A. Poetzsch-Heffter
Mitarbeit: J. Meyer, P. Müller
Aktualisiert: J. Knoop, M. Müller-Olm, U. Scheben, D. Keller, A. Thies,
J. Hagemann, M. Paap

63611
Einführung in die objektorientierte Programmierung

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Inhaltsverzeichnis

Hinweise zur Bearbeitung des Lehrtextes **IX**

Lektion 1 (Kapitel 1) **1**

Studierhinweise zur Lektion 1 **1**

1 Objektorientierung: Ein Einstieg **3**
1.1 Objektorientierung: Konzepte und Stärken 3
1.1.1 Gedankliche Konzepte der Objektorientierung 4
1.1.2 Abgrenzung zur prozeduralen Programmierung 9
1.1.2.1 Prozedurale Programmierung 9
1.1.2.2 Objektorientierte Programmierung 10
1.1.3 Objektorientierung als Antwort 13
1.2 Programmiersprachlicher Hintergrund 18
1.2.1 Grundlegende Sprachmittel am Beispiel von Java 18
1.2.1.1 Objekte und Werte: Eine begriffliche Abgrenzung 18
1.2.1.2 Werte, Typen und Variablen in Java 20
1.2.1.3 Arrays in Java 22
1.2.1.4 Operation, Zuweisungen und Auswertung in Java 23
1.2.1.5 Ausführung eines Java-Programms 27
1.2.1.6 Anweisungen, Blöcke und deren Ausführung . 29
1.2.1.7 Klassische Kontrollstrukturen. 31
1.2.1.8 Abfangen von Ausnahmen 35
1.2.1.9 Ausnahmebehandlung und Nachrichtenversand 38
1.2.2 Objektorientierte Programmierung mit Java 40
1.2.2.1 Objekte, Klassen, Methoden, Konstruktoren . . 40
1.2.2.2 Spezialisierung und Vererbung 42
1.2.2.3 Subtyping und dynamisches Binden 44
1.2.3 Objektorientierte Sprachen im Überblick 46

Musterlösungen der Ad-hoc-Aufgaben der Lektion 1 **49**

Selbsttestaufgaben zur Lektion 1 **55**

Musterlösungen der Selbsttestaufgaben zur Lektion 1 **59**

Lektion 2 (Kapitel 2 - 4) **67**

Studierhinweise zur Lektion 2 **67**

2 Objekte, Klassen, Kapselung **69**
2.1 Objekte und Klassen 69
2.1.1 Beschreibung von Objekten 69
2.1.2 Klassen beschreiben Objekte 71
2.1.2.1 Klassendeklarationen, Attribute und Objekte . 73
2.1.2.2 Methodendeklaration 74
2.1.2.3 Konstruktordeklaration 75
2.1.2.4 Objekterzeugung, Attributzugriff und Methodenaufruf 75
2.1.2.5 Objektorientierte Programme 78
2.1.3 Weiterführende Sprachkonstrukte 80
2.1.3.1 Initialisierung und Überladen 80
2.1.3.2 Klassenmethoden und Klassenattribute 83
2.1.4 Rekursive Klassendeklaration 90
2.2 Kapselung und Strukturierung von Klassen 93
2.2.1 Kapselung und Schnittstellenbildung: Erste Schritte . . . 93
2.2.2 Strukturieren von Klassen 95
2.2.2.1 Innere Klassen 96
2.2.2.2 Strukturierung von Programmen: Pakete 103
2.2.3 Beziehungen zwischen Klassen 111

3 Typisierung und Subtyping **115**
3.1 Zielsetzung von Typisierung 115
3.2 Nachteile von Typisierung 117
3.3 Ein allgemeinster Typ für Objekte 118
3.4 Basisdatentypen und Subtyping 122
3.5 Subtyping bei Arrays 124

4 Klassifizierung von Objekten **127**
4.1 Klassifikation 127
4.2 Spezialisierung und Abstraktion 129
4.2.1 Spezialisierung: Vererbung und Überschreiben 129
4.2.2 Abstraktion: Schnittstellentypen (Interfaces) 132
4.2.3 Zusammenfassung 134

Musterlösungen der Ad-hoc-Aufgaben der Lektion 2 **135**

Selbsttestaufgaben zur Lektion 2 **143**

Musterlösungen der Selbsttestaufgaben zur Lektion 2 **147**

Lektion 3 (Kapitel 5 - 7) **153**

Studierhinweise zur Lektion 3 **153**

5 Subtyping genauer betrachtet **155**
5.1 Interfacetypen und Subtyping 155
5.1.1 Deklaration von Interfacetypen 156
5.1.2 Deklaration von Subtyping 157
5.1.2.1 Interfacetypen als Subtypen 157
5.1.2.2 Klassentypen als Subtypen 158
5.1.3 Klassifikation und Subtyping 160
5.1.4 Typhierarchien erweitern 161
5.1.5 Was es heißt, ein Subtyp zu sein 162
5.1.5.1 Syntaktische Bedingungen 162
5.1.5.2 Konformes Verhalten 166

6 Vererbung genauer betrachtet **167**
6.1 Anpassung geerbter Methoden 167
6.2 Vererbung und Objektinitialisierung 169
6.3 Vererbung und innere Klassen 173
6.4 Programmierung für Vererbung 175
6.5 Vererbung, Subtyping und Subclassing 176
6.5.1 Abstrakte Klassen und Methoden 178
6.5.2 Mehrfachvererbung 179
6.6 Vererbung und Kapselung 181
6.6.1 Kapselungskonstrukte im Zusammenhang mit Vererbung 181
6.6.2 Zusammenspiel von Vererbung und Kapselung 184
6.6.3 Realisierung gekapselter Objektgeflechte 186
6.7 Verstecken vs. Überschreiben 196
6.8 Auflösen von Methodenaufrufen 198
6.9 OO-Programmierung und Wiederverwendung 201

7 Aufzählungstypen in Java **203**

Musterlösungen der Ad-hoc-Aufgaben der Lektion 3 **207**

Selbsttestaufgaben zur Lektion 3 **213**

Musterlösungen der Selbsttestaufgaben zur Lektion 3 **217**

Lektion 4 (Kapitel 8 - 10) **221**

Studierhinweise zur Lektion 4 **221**

8 Parametrisierung von Typen **223**
8.1 Parametrisierung von Klassen 225
8.2 Besonderheiten parametrischer Klassen 230
8.2.1 Klassenattribute und -methoden im Kontext parametrischer Klassen 230
8.2.2 Überladen von Methodennamen im Kontext parametrischer Klassen 232
8.3 Parametrische Klassen mit inneren Klassen 233
8.4 Beschränkt parametrische Klassen 235
8.5 Subtyping im Kontext parametrischer Typen 239
8.5.1 Deklaration, Erweiterung und Implementierung parametrischer Schnittstellen 239
8.5.2 Die Interfaces Iterable, Iterator und Comparable 241
8.5.3 Subtyping bei parametrischen Behältertypen 243

9 Polymorphie **247**
9.1 Subtyp-Polymorphie 247
9.2 Parametrische Polymorphie 248
9.3 Beschränkt parametrische Polymorphie 248
9.4 Ad-hoc-Polymorphie 248

10 Bausteine für objektorientierte Programme **251**
10.1 Bausteine und Bibliotheken 251
10.1.1 Bausteine in der Programmierung 251
10.1.1.1 Beziehungen zwischen Bausteinen 252
10.1.1.2 Komposition von Bausteinen 253
10.1.1.3 Beschreibung von Bausteinen 254
10.1.2 Überblick über die Java-Bibliothek 254
10.2 Ausnahmebehandlung mit Bausteinen 257
10.2.1 Eine Hierarchie von einfachen Bausteinen 257
10.2.2 Zusammenspiel von Sprache und Bibliothek 259
10.3 Ströme: Bausteine zur Ein- und Ausgabe 262
10.3.1 Ströme: Eine Einführung 262
10.3.2 Ein Baukasten mit Stromklassen 266
10.3.2.1 Javas Stromklassen: Eine Übersicht 267
10.3.2.2 Ströme von Objekten 272

Musterlösungen der Ad-hoc-Aufgaben der Lektion 4 **277**

Selbsttestaufgaben zur Lektion 4 **279**

Musterlösungen der Selbsttestaufgaben zur Lektion 4 **283**

Lektion 5 (Kapitel 11 - 12) **289**

Studierhinweise zur Lektion 5 **289**

11 Rückruffunktionen und Beobachter **291**
11.1 Rückruffunktionen . . . 291
11.1.1 Interfaces zur Simulation von Methodenreferenzen . . . 292
11.1.2 Lokale Klassen . . . 295
11.1.3 Anonyme Klassen . . . 296
11.1.4 Lambda-Ausdrücke . . . 297
11.2 Das Beobachter-Entwurfsmuster . . . 299

12 Objektorientierte Programmgerüste **305**
12.1 Frameworks - Eine kurze Einführung . . . 306
12.1.1 Framework: Ja oder nein? . . . 307
12.1.2 Unvollständigkeit und Modellierung von Frameworks . 308
12.2 Ein Framework für Bedienoberflächen: AWT . . . 309
12.2.1 Aufgaben und Aufbau grafischer Bedienoberflächen . . 309
12.2.2 Die Struktur des Abstract Window Toolkit . . . 311
12.2.2.1 Das abstrakte GUI-Modell des AWT . . . 311
12.2.2.2 Komponenten . . . 312
12.2.2.3 Darstellung . . . 315
12.2.2.4 Ereignissteuerung . . . 317
12.2.2.5 Programmtechnische Realisierung des AWT im Überblick . . . 320
12.2.3 Praktische Einführung in das AWT . . . 321
12.2.3.1 Initialisieren und Anzeigen von Hauptfenstern 322
12.2.3.2 Behandeln von Ereignissen . . . 323
12.2.3.3 Elementare Komponenten . . . 326
12.2.3.4 Komponentendarstellung selbst bestimmen . . 328
12.2.3.5 Layout-Manager: Anordnen von Komponenten 331
12.2.3.6 Erweitern des AWT . . . 337
12.2.4 Rückblick auf die Einführung ins AWT . . . 342
12.3 Anwendung von Frameworks . . . 343
12.3.1 Frameworks und Software-Architekturen . . . 343
12.3.2 Entwicklung grafischer Bedienoberflächen . . . 346
12.3.2.1 Anforderungen . . . 347
12.3.2.2 Analyse der Dialogführung . . . 347
12.3.2.3 Entwurf des Datenmodells . . . 348
12.3.2.4 Entwicklung der Darstellung . . . 350
12.3.2.5 Realisierung der Steuerung . . . 353
12.3.2.6 Verbinden der Teilsysteme . . . 354

Musterlösungen der Ad-hoc-Aufgaben der Lektion 5 **357**

Selbsttestaufgaben zur Lektion 5 **363**

Musterlösungen der Selbsttestaufgaben zur Lektion 5 **367**

Lektion 6 (Kapitel 13) **373**

Studierhinweise zur Lektion 6 **373**

13 Parallelität **375**
13.1 Parallelität und Objektorientierung 375
13.1.1 Allgemeine Aspekte von Parallelität 376
13.1.2 Parallelität in objektorientierten Sprachen 378
13.2 Lokale Parallelität in Java-Programmen 379
13.2.1 Java-Threads 379
13.2.1.1 Programmtechnische Realisierung von Threads in Java 379
13.2.1.2 Benutzung von Threads 383
13.2.2 Synchronisation 393
13.2.2.1 Synchronisation: Problemquellen 393
13.2.2.2 Ein objektorientiertes Monitorkonzept 398
13.2.2.3 Synchronisation mit Monitoren 403
13.2.3 Zusammenfassung der sprachlichen Umsetzung von lokaler Parallelität 413

Musterlösungen der Ad-hoc-Aufgaben der Lektion 6 **415**

Selbsttestaufgaben zur Lektion 6 **417**

Musterlösungen der Selbsttestaufgaben zur Lektion 6 **423**

Lektion 7 (Kapitel 14) **431**

Studierhinweise zur Lektion 7 **431**

14 Programmierung verteilter Objekte **433**
14.1 Verteilte objektorientierte Systeme 433
14.1.1 Grundlegende Aspekte verteilter Systeme 433
14.1.2 Programmierung verteilter objektorientierter Systeme . 436
14.2 Client-Server-Kommunikation über Sockets 438
14.2.1 Sockets: Allgemeine Eigenschaften 439
14.2.2 Realisierung eines einfachen Servers 440
14.2.3 Realisierung eines einfachen Clients 443
14.2.4 Server mit mehreren Threads 445
14.2.5 Client und Server im Internet 447
14.2.5.1 Dienste im Internet 447
14.2.5.2 Zustandslose Protokolle 448
14.2.5.3 Sitzungen über zustandsbehaftete Protokolle . 448
14.2.5.4 Sitzungen über zustandslose Protokolle 449
14.2.5.5 Praktisches Beispiel 450
14.3 Kommunikation über entfernten Methodenaufruf 454
14.3.1 Problematik entfernter Methodenaufrufe 454
14.3.1.1 Behandlung verteilter Objekte 454
14.3.1.2 Simulation entfernter Methodenaufrufe über Sockets 458
14.3.2 Realisierung entfernter Methodenaufrufe in Java 459
14.3.2.1 Der Stub-Skeleton-Mechanismus 459
14.3.2.2 Entfernter Methodenaufruf in Java 460

Musterlösungen der Ad-hoc-Aufgaben der Lektion 7 **469**

Selbsttestaufgaben zur Lektion 7 **473**

Musterlösungen der Selbsttestaufgaben zur Lektion 7 **475**

Verzeichnis Online-Quellen **481**

Literaturverzeichnis **483**

Stichwortverzeichnis **485**

Hinweise zur Bearbeitung des Lehrtextes

Einleitung Die objektorientierte Programmierung modelliert und realisiert Software-Systeme als „Populationen“ kooperierender Objekte. Vom Prinzip her ist sie demnach eine Vorgehensweise, um Programme gemäß einem bestimmten Grundmodell zu entwerfen und zu strukturieren. In der Praxis ist sie allerdings – wie jede Form der Programmierung – eng mit dem Studium und der Verwendung geeigneter Programmiersprachen verbunden.

Der Lehrtext verfolgt eine zweifache Zielsetzung: Zum einen will er Ihnen die Grundbegriffe, Konzepte und Denkweisen der objektorientierten Programmierung vermitteln. Zum Anderen soll er Ihnen einen praktischen Einstieg in eine der heute wohl wichtigsten objektorientierten Programmiersprachen – Java – ermöglichen, auf dessen Basis Sie sich weitere Kenntnisse bei Bedarf selbst aneignen können.

Warum Java? Im Vergleich zu anderen objektorientierten Sprachen bietet die Abstützung und Konzentration auf Java einige Vorteile: Java wurde als objektorientierte Sprache entwickelt – ist also keine Erweiterung einer prozeduralen Sprache – so dass es relativ frei von Erblasten ist, welche von den objektorientierten Aspekten ablenken würden. Die meisten objektorientierten Konzepte lassen sich daher in Java recht gut veranschaulichen. Java besitzt ein relativ sauberes Typsystem, was zum einen die Programmierung erleichtert und zum anderen eine gute Grundlage ist, um wichtige Konzepte wie z. B. Subtyping zu behandeln. Weitere Vorteile sind die umfangreiche standardisierte Klassenbibliothek und die freie Verfügbarkeit von Entwicklungswerkzeugen, von einfachen Kommandozeilenprogrammen bis hin zu ausgereiften integrierten Entwicklungsumgebungen.

Vollständigkeit / Aktualität Trotz der Wahl der Sprache Java ist der Lehrtext *kein* klassischer „Java-Kurs“. Insbesondere strebt er keine Vollständigkeit bzgl. der – seit geraumer Zeit mit jeder Hauptversion mehr werdenden – Sprachmittel der Sprache Java an, sondern beschränkt sich auf eine (immer noch umfangreiche) Teilmenge, bei deren Auswahl sowohl konzeptionelle als auch praktische Gesichtspunkte berücksichtigt wurden.

Ähnliches gilt für die Java-Standardklassenbibliothek: Bei vielem, was im Lehrtext vermittelt wird, handelt es sich um grundlegende Vorgehensweisen, die sich in ähnlicher Form auch in anderen objektorientierten Sprachen

wiederfinden. Eine Behandlung der jeweils neuesten Umsetzung in der Java-Standardklassenbibliothek bedürfte nicht nur ständiger Aktualisierung, sondern würde auch zwangsläufig die allgemeinen Grundlagen hinter den Details der immer spezieller werdenden Umsetzung zurücktreten lassen.

Die beiden auffälligsten Beispiele für diese bewusste „Nicht-Aktualität“ im Lehrtext sind wahrscheinlich die Kapitel über grafische Benutzeroberflächen und über lokale Parallelität, weswegen wir auf diese kurz eingehen wollen:

Für die Modellierung grafischer Benutzeroberflächen verwendet dieser Lehrtext das in seinen Fähigkeiten recht eingeschränkte Framework AWT und nicht, wie Sie vielleicht erwartet hätten, Swing oder JavaFX. Sollten Sie sich später mit einem dieser beiden aktuelleren (aber auch wesentlich umfangreicheren und komplexeren) Frameworks beschäftigen, werden Sie allerdings feststellen, dass Sie nichts „umsonst gelernt“ haben, da die grundlegenden Konzepte sehr ähnlich sind, und ein Umstieg durch „Hinzulernen“ relativ einfach möglich ist.

Etwas anders gelagert ist der Fall der „lokalen Parallelität“: Der Lehrtext beschreibt dort Vorgehensweisen, die inzwischen zum Teil in Form von Halbfertigbausteinen in Javas Standard-Klassenbibliothek integriert wurden, also oft gar nicht mehr in der im Lehrtext gezeigten Weise von Hand ausprogrammiert werden müssen. Allerdings wird man diese Bausteine nur sinnvoll einsetzen können, wenn man die von ihnen adressierten Probleme überhaupt versteht. Und dazu ist die Beschäftigung mit der problemnahen „händischen“ Vorgehensweise ausgesprochen sinnvoll. Außerdem hat dies den Vorteil, anschließend auch Lösungen der gleichen Probleme in anderen Programmiersprachen als Java leichter nachvollziehen zu können.

Online-Quellen Neben den üblichen Verweisen auf das Literaturverzeichnis finden sich im Lehrtext an etlichen Stellen Verweise auf Onlinequellen, erkennbar am Symbol $\rightsquigarrow$. Um diese einfacher aktuell halten zu können, wurden sie – analog zu den Literaturhinweisen – am Ende des Lehrtextes in einem separaten Verzeichnis gesammelt. In der PDF-Version des Lehrtextes finden Sie dort anklickbare Verlinkungen zu den betreffenden Webseiten. Falls diese einmal nicht mehr funktionieren, sollten sich die betreffenden Inhalte meistens mit Hilfe des Titels und/oder der angegebenen Suchbegriffe dennoch auffinden lassen.

Entwicklungsumgebung Sollten Sie bereits Erfahrung mit Java haben und sich entsprechende Entwicklungswerkzeuge installiert haben, können Sie diese natürlich auch für die Bearbeitung des Lehrtextes verwenden. Ansonsten empfehlen wir Ihnen die Verwendung der Integrierten Entwicklungsumgebung (IDE) Eclipse. Die Hinweise zur Installation und zu ersten Schritten mit der IDE haben wir in einen „Vorkurs“ ausgelagert, um sie besser aktuell halten zu können. Sie finden den Vorkurs in der Moodle-Umgebung dieses Moduls.

Hilfe bei der Bearbeitung des Lehrtextes Das Lehrgebiet „Programmiersysteme“ nutzt zur Betreuung des Moduls 63611 ein Diskussionsforum auf der Moodle-Seite des Moduls. Dieses ist sowohl für organisatorische Informationen des Lehrgebiets gedacht als auch für Ihre Fragen inhaltlicher Art. Wir empfehlen dringend, dieses Forum zu abonnieren und intensiv zu nutzen.

Selbsttestaufgaben, Ad-hoc-Aufgaben Sie finden am Ende jeder Lektion Selbsttestaufgaben, die Ihnen zusätzlich zu den Einsendeaufgaben ermöglichen sollen, selbst zu überprüfen, inwieweit Sie den Stoff verstanden haben bzw. wo es evtl. noch Probleme gibt. Diese Selbsttestaufgaben sind dazu gedacht, nach der Bearbeitung der entsprechenden Lektion gelöst zu werden. Darüber hinaus finden Sie auch innerhalb der einzelnen Lektionen weitere Aufgaben kleineren Umfangs. Diese Ad-hoc-Aufgaben lösen Sie am besten direkt während der Bearbeitung der Lektion, sobald sie im Text auftauchen. Die Lösungen für die Ad-hoc-Aufgaben finden Sie ebenso wie die der Selbsttestaufgaben am Ende der betreffenden Lektion. Sowohl die Selbsttestaufgaben als auch die Ad-hoc-Aufgaben sind als Teil des Lehrtextes zu verstehen. In Aufgabenstellung und/oder Musterlösung vermittelte Informationen sind also ggf. Teil des (klausurrelevanten) Stoffes.

Codebeispiele Im Lehrtext und in den Aufgaben findet sich eine Vielzahl von Codebeispielen. Diese sollten Sie nicht nur lesen, sondern aktiv ausprobieren und um eigene Ideen ergänzen. Auch Fragen zu dabei auftauchenden Problemen sind übrigens im Diskussionsforum willkommen! Generell sollten Sie bei der Bearbeitung des Lehrtextes möglichst viel aktiv programmieren. Das ist zwar mit Zeitaufwand verbunden, aber hier „sparen“ zu wollen, ist schon alleine deshalb nicht sinnvoll, weil die meisten von Ihnen in ihrem Studiengang auch ein Programmierpraktikum absolvieren werden. Spätestens dabei wird sich jede Minute mehrfach auszahlen, in der Sie sich bereits praktisch mit Java beschäftigt haben.

Bitte beachten Sie, dass die Beispiele teilweise lediglich der Illustration eines Konzepts dienen und nicht immer eine in der Realität sinnvolle Herangehensweise im Rahmen des im Beispiel verwendeten Themas darstellen. Wenn z. B. die Datenkapselung mit Hilfe von Paketen anhand unterschiedlicher Paketzugehörigkeiten von Mitarbeitenden eines Unternehmens illustriert wird, dann heißt das nicht, dass in real existierenden Unternehmen das Rechtemanagement mit Hilfe von Paketen realisiert würde oder dass dies eine gute Idee wäre.

Studierhinweise zur Lektion 1

Diese Lektion beschäftigt sich mit dem ersten Kapitel des Lehrtextes. Sie sollten dieses Kapitel im Detail studieren und verstehen. Nehmen Sie sich insbesondere die Zeit, die Sprachkonstrukte an kleinen, selbst entworfenen Beispielen im Rahmen dieser Lektion zu üben! Ein bloßes Durchlesen des Textes ist nicht ausreichend.

Lernziele:

- Grundlegende Konzepte der objektorientierten Programmierung
- Abgrenzung zwischen objektorientierter Programmierung und prozeduraler Programmierung
- Sprachliche Grundlagen der Programmierung mit Java
- Programmtechnische Fähigkeiten im Umgang mit Java

Kapitel 1

Objektorientierung: Ein Einstieg

Zentrales Ziel dieses Kapitels ist es, das gedankliche Modell, welches der objektorientierten Programmierung zugrunde liegt, systematisch herauszuarbeiten.

Außerdem bietet das Kapitel eine kurze Einführung in die programmiersprachliche Umsetzung objektorientierter Konzepte in Java und erläutert einige sprachliche Grundlagen, die im Rest des Lehrtextes benötigt werden.

1.1 Objektorientierung: Konzepte und Stärken

Dieser Abschnitt bietet eine erste Einführung in objektorientierte Konzepte (Abschn. 1.1.1), grenzt die objektorientierte Programmierung gegenüber der prozeduralen Programmierung ab (Abschn. 1.1.2) und untersucht die Rolle der objektorientierten Programmierung als Antwort auf bestimmte softwaretechnische Anforderungen (Abschn. 1.1.3).

Zunächst wollen wir allerdings kurz den Begriff „Objektorientierte Programmierung“ reflektieren. Dabei soll insbesondere deutlich werden, dass objektorientierte Programmierung mehr ist als die Programmierung in einer objektorientierten Programmiersprache.

Objektorientierte Programmierung: Was bedeutet das? Der Begriff „Programmierung“ wird mit unterschiedlicher Bedeutung verwendet. Im engeren Sinne wird darunter das Aufschreiben eines Programms in einer gegebenen Programmiersprache verstanden: Wir sehen Programmierende vor uns, die einen Programmtext in ihrem Rechner editieren. Im weiteren Sinn ist die Entwicklung und Realisierung von Programmen ausgehend von einem allgemeinen Softwareentwurf gemeint, d. h. einem Softwareentwurf, in dem noch keine programmiersprachspezifischen Entscheidungen getroffen sind. Programmierung in diesem Sinne beschäftigt sich also auch mit Konzepten und Techniken zur Überwindung der Kluft zwischen Softwareentwurf und Programmen. *In diesem Lehrtext wird Programmierung in dem weiter gefassten Sinn verstanden.*

Programmierung

Objektorientierte Programmierung ist demnach Programmentwicklung mit Hilfe objektorientierter Konzepte und Techniken. Dabei spielen naturgemäß programmiersprachliche Aspekte eine zentrale Rolle. Im Gesamt-

Obj.-or. Programmierung

bild der Softwareentwicklung wird die objektorientierte Programmierung durch objektorientierte Techniken für Analyse, Entwurf und Testen ergänzt. Die Grundkonzepte der Programmierung beeinflussen dabei sowohl Programmiersprachen und -techniken als auch den Softwareentwurf und umgekehrt. Resultat einer objektorientierten Programmentwicklung sind in der Regel Programme, die in einer objektorientierten Programmiersprache verfasst sind.

Um in diesem umfassenden Sinne objektorientiert programmieren zu können, reicht es nicht, bestimmte Techniken zu beherrschen, man muss vielmehr lernen, „objektorientiert zu denken". Erfahrungsgemäß fällt manchen Studierenden dabei insbesondere der Umstieg von der imperativ-prozeduralen Denkweise – wie sie z. B. im Modul 63811 anhand einer Pascal-ähnlichen Programmiersprache vermittelt wird – zur objektorientierten Denkweise nicht ganz leicht, obwohl Modul 63811 versucht, den Zugang zur Objektorientierung nicht zu verstellen.

Wir werden deshalb nach einer ersten Einführung in die gedanklichen Konzepte der objektorientierten Programmierung noch einmal kurz zusammenfassen, wie Informationen und deren Verarbeitung in der prozeduralen Programmierung modelliert werden und auf die wesentlichen Unterschiede zur objektorientierten Programmierung hinweisen (siehe Abschn. 1.1.2).

1.1.1 Gedankliche Konzepte der Objektorientierung

Die Objektorientierung bezieht ihre gedanklichen Grundlagen aus Vorgängen der realen Welt. Das ihr zugrundeliegende Weltbild ist das von Objekten, die jeweils eine Identität haben, die einander *Nachrichten* schicken und die als Reaktion auf Nachrichten ihren eigenen Zustand verändern können. Welche Nachrichten ein Objekt verstehen kann, zählt zu seinen Eigenschaften. Wie ein Objekt auf den Empfang einer bestimmten Nachricht reagiert, fällt dabei in die Zuständigkeit des Objekts selbst. Objekte haben zudem eine Lebensdauer. Sie können entstehen und wieder vergehen – das objektorientierte Weltbild ist also in vielerlei Hinsicht dynamisch.

Nachrichten

Damit ein Objekt gezielt einem anderen eine Nachricht schicken kann, muss es das andere kennen. Ein Objekt kennt ein anderes, indem es als Teil seines eigenen Zustands eine sog. *Referenz* auf das andere Objekt besitzt. Wie diese Referenz technisch realisiert ist (z. B. als Zeiger), ist für unsere Zwecke relativ unwichtig. Es reicht zu wissen, dass eine Referenz es erlaubt, über sie ein Objekt anzusprechen.

Referenz

Welche anderen Objekte ein Objekt kennen kann, zählt zu seinen Eigenschaften, welche es tatsächlich kennt, macht den Zustand eines Objektes aus und unterliegt mit diesem der Veränderung. Um neue Bekanntschaften zu schließen, können einem Objekt Referenzen auf ein oder mehrere andere Objekte als Parameter einer Nachricht geschickt werden. Der Empfang einer Nachricht durch ein Objekt führt in der Regel zum Versand weiterer Nachrichten durch das empfangende Objekt sowohl an andere Objekte als auch an sich selbst. Auch das Entstehen und Vergehen von Objekten erfolgt in der Regel als Reaktion auf den Empfang einer Nachricht. Objekte sind grundsätzlich

selbständige Ausführungseinheiten, die unabhängig voneinander und parallel arbeiten können[1].

Da Objekte so allgemeine Dinge wie Personen oder Dokumente, aber auch so spezielle Dinge wie Zahlen oder Wahrheitswerte sein können und Nachrichten so allgemeine wie „Schreib dich ein“ oder „Drucke dich aus“, aber auch so spezielle wie „+“ oder „-“, hat man damit im Prinzip schon fast alles, was man zum Programmieren braucht. Die einzigen zusätzlich benötigten Konzepte sind das der Variable und der Wertzuweisung, die Sie vermutlich bereits aus anderen Programmiersprachen, zumindest aber aus der Mathematik kennen.

Identität Objekte verhalten sich in mehreren Aspekten wie Gegenstände der materiellen Welt (bei Gegenständen denke man etwa an Autos, Lampen, Telefone, Lebewesen etc.). Insbesondere haben sie eine *Identität* und einen Aufenthaltsort: Ein Objekt kann nicht an zwei Orten gleichzeitig sein. Es kann sich ändern, bleibt dabei aber dasselbe Objekt (man denke beispielsweise daran, dass ein Auto umlackiert werden kann, ohne dass sich dabei seine Identität ändert, oder dass bei einem Menschen im Laufe seines Lebens fast alle Zellen ausgetauscht werden, die Identität des Menschen davon aber unberührt bleibt). Objekte im Sinne der objektorientierten Programmierung unterscheiden sich also von üblichen mathematischen Objekten wie Zahlen, Funktionen, Mengen, usw. (Zahlen haben keine Lebensdauer, keinen „Aufenthaltsort“ und keinen Zustand.)

Identität

Nachrichten und Methoden. Ein zentraler Aspekt der Objektorientierung, der sich direkt aus dem beschriebenen Modell ergibt, ist eine klare Trennung von Auftragserteilung und Auftragsdurchführung. Betrachten wir dazu ein kleines Beispiel: Wir nehmen an, dass eine Frau K. ein Buch kaufen möchte. Um das Buch zu besorgen, erteilt Frau K. der Buchhandlung Mississippi den Auftrag, das Buch zu beschaffen und es ihr zuzuschicken. Die Auftragsdurchführung liegt dann in der Hand der Buchhandlung. Genauer besehen passiert Folgendes:

1. Frau K. löst eine Aktion aus, indem sie der Buchhandlung einen Auftrag gibt. Übersetzt in die Sprache der Objektorientierung heißt das, dass ein Senderobjekt, nämlich Frau K., einem Empfängerobjekt, nämlich der Buchhandlung, eine *Nachricht* schickt. Diese Nachricht besteht üblicherweise aus der Bezeichnung des Auftrags (Buch beschaffen und zuschicken) und weiteren Parametern (etwa dem Buchtitel).

   Nachricht

2. Die Buchhandlung besitzt eine bestimmte *Methode*, die sie befähigt, die an sie gesendete Nachricht zu verarbeiten. In der Methode ist festgelegt, *wie* sie Frau K.'s Auftrag durchführt (etwa: nachschauen, ob Buch am Lager, ansonsten billigsten Großhändler suchen etc.). Diese Festlegung erfolgt technisch in Form einer Folge von *Anweisungen* einer Programmiersprache, die bei Ausführung der Methode sequentiell abgearbeitet

   Methode

[1] Dieses Modell wird in vielen in der Praxis eingesetzten Programmiersprachen nicht umgesetzt, siehe auch Absatz „Relativierung inhärenter Parallelität“ auf Seite 12

> wird, eine Vorgehensweise, die Ihnen von den Funktionen und Prozeduren der imperativ-prozeduralen Programmierung bekannt sein dürfte. Dieses „Wie“, also den Inhalt der Methode, braucht Frau K. nicht zu kennen. Auch wird die Methode, wie das Buch beschafft wird, von Buchhandlung zu Buchhandlung im Allgemeinen verschieden sein.

Die konzeptionelle Trennung von Auftragserteilung und Auftragsdurchführung, d. h. die Unterscheidung von Nachricht und Methode, führt zu einer klaren Aufgabenteilung: Die Auftraggebende muss sich jemanden suchen, der ihren Auftrag versteht und durchführen kann. Sie weiß im Allgemeinen nicht, wie der Auftrag bearbeitet wird (Geheimnisprinzip, engl. Information Hiding). Der Auftragsempfänger ist für die Durchführung verantwortlich und besitzt dafür eine Methode.

Nachrichtenversand und Zustandsänderung Wenn wir einem Objekt eine Nachricht senden, verfolgen wir damit normalerweise eine bestimmte Absicht. Im Fall unserer Frau K. besteht diese darin, von einer Buchhandlung ein bestimmtes Buch zu erhalten. Dabei dürfte es für sie zweitrangig sein, um welches Objekt es sich bei der Buchhandlung genau handelt, solange dieses nur eine Buchhandlung und damit in der Lage ist, ihre Nachricht erfolgreich zu verarbeiten.

In vielen Fällen bezieht sich die Absicht des Nachrichtenversenders aber auch auf ein ganz bestimmtes Objekt. Betrachten wir dazu den Fall, dass Frau K. ihr Motorrad starten möchte. Um dies zu tun, schickt sie dem Motorrad eine entsprechende Nachricht. Das Motorrad verarbeitet diese mit seiner dafür vorgesehenen Methode und ändert seinen Zustand: Der Motor läuft. In diesem Fall ist es für Frau K. sicherlich nicht egal, *welchem* Motorrad sie ihre Nachricht sendet, schließlich möchte sie den Zustand *ihres* Motorrades ändern.

Objektzustand

Wie wir gesehen haben, befindet sich die Fähigkeit, Nachrichten zu verarbeiten, in den Methoden des Objekts, das die Rolle des Nachrichtenempfängers spielt. Bleibt die Frage, wo sich der Zustand eines Objekts befindet. Wie schon bei den Methoden ist auch hier die Antwort etwas, das sie bereits von der imperativ-prozeduralen Programmierung her kennen: Ein Objekt verfügt typischerweise über eine Reihe von lokalen *Variablen*, deren Werte in ihrer Gesamtheit den Zustand des Objekts bilden. Man bezeichnet diese objektlokalen Variablen als die Attribute oder auch Instanzvariablen[2] des betreffenden Objekts.

Attribut

Klassifikation und Vererbung. Die zentrale Rolle, welche das Versenden und Empfangen von Nachrichten in der objektorientierten Programmierung spielt, legt es nahe, Objekte unter dem Aspekt zu gruppieren, welche Nachrichten sie verstehen. Eine solche „Klassifikation“ von Gegenständen und Begriffen ist dabei auch in der realen Welt gängig und durchzieht beinahe alle Bereiche unseres Lebens. Beispielsweise sind Händler und Geschäfte nach

[2] Der Grund für diese Bezeichnung wird später noch erläutert.

Branchen klassifiziert. Jede Branche ist dabei durch die Dienstleistungen charakterisiert, die ihre Händler erbringen: Buchhandlungen handeln mit Büchern, Lebensmittelhändler mit Lebensmitteln, entsprechend verstehen sie unterschiedliche Nachrichten und unsere Frau K. ist gut beraten, sich mit ihrem Auftrag zu Beschaffung eines Buchs nicht an einen Lebensmittelhändler zu wenden.

Was im Geschäftsleben die Branchen sind, sind in den meisten objektorientierten Sprachen die Klassen. Eine *Klasse* legt die Fähigkeiten und Eigenschaften fest, die allen ihren Objekten gemeinsam sind. Klassen lassen sich hierarchisch organisieren. Abbildung 1.1 demonstriert eine solche hierarchische *Klassifikation* am Beispiel von Branchen. Dabei besitzen die übergeordne-

Klasse

Klassifikation

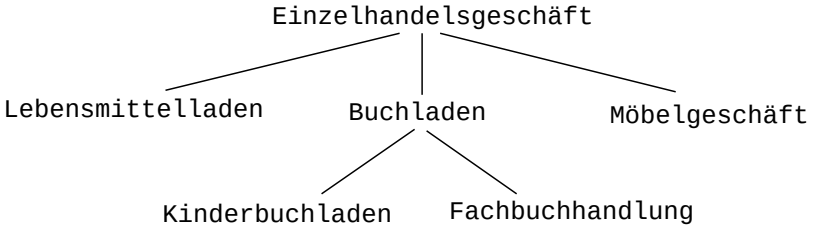

Abbildung 1.1: Klassifikation von Geschäften

ten Klassen (Superklassen) nur Eigenschaften, die den untergeordneten Klassen (Subklassen) bzw. ihren Objekten gemeinsam sind. Für die Beschreibung der Eigenschaften von Klassen und Objekten bringt die hierarchische Organisation drei entscheidende Vorteile gegenüber einer unstrukturierten Menge von Klassen:

1. Es lassen sich *abstrakte* Klassen bilden. Das sind Klassen, die selbst keine direkten Objekte haben, sondern nur dafür angelegt sind, Gemeinsamkeiten ihrer Subklassen zusammenzufassen. Jedes Objekt, das einer abstrakten Klasse zugerechnet wird, ist also immer ein Objekt einer konkreten Subklasse der abstrakten Klasse. Beispielsweise ist „Einzelhandelsgeschäft“ eine abstrakte Klasse. Sie fasst die Eigenschaften zusammen, die allen Geschäften gemeinsam sind. Es gibt aber kein Geschäft, das nur ein Einzelhandelsgeschäft ist und keiner Branche zugeordnet werden kann. Anders ist es mit der Klasse Buchladen: Es gibt Buchläden mit einem gemischten Angebot, die zu keiner *spezielleren Klasse* gehören, d. h. die Klasse Buchladen ist nicht abstrakt.

   abstrakte Klassen

   Die Entscheidung, ob eine Klasse als abstrakt anzusehen ist, kann dabei je nach Kontext durchaus unterschiedlich ausfallen. Beispielsweise könnte eine Klasse Hund im Kontext einer statistischen Erhebung zur Haltung von Haustieren eine konkrete Klasse sein. Ein Hund wäre dann einfach ein Hund. Im Kontext eines Hundesalons hingegen, in dem die speziellen Eigenschaften verschiedener Hunderassen relevant sind, wäre Hund eine abstrakte Klasse. In einem solchen Kontext gäbe es dann keine Hunde, die *nur* Hunde sind: Jeder Hund wäre immer auch ein Objekt einer der Subklassen von Hund wie etwa Pudel, Dackel oder Schäferhund.

2. Eigenschaften und Methoden, die mehreren Klassen gemeinsam sind, brauchen nur einmal bei der übergeordneten Klasse beschrieben zu werden und können von deren Subklassen *geerbt* werden. Beispielsweise besitzt jedes Einzelhandelsgeschäft eine Auftragsabwicklung. Die Standardverfahren der Auftragsabwicklung, die bei allen Geschäften gleich sind, brauchen nur einmal bei der Klasse Einzelhandelsgeschäft beschrieben zu werden. Alle Subklassen können diese Verfahren erben und an ihre speziellen Verhältnisse anpassen. Für derartige Spezialisierungen stellt die objektorientierte Programmierung bestimmte Techniken zur Verfügung.

Vererbung

Spezialisierung

3. Durch die Existenz einer Klassenhierarchie und des beschriebenen Mechanismus der Vererbung von Methoden innerhalb dieser Klassenhierarchie kann man Objekte, welche einer Klasse K angehören, ohne Weiteres auch als Objekte aller Superklassen von K ansehen. Schließlich erbt die spezielle Klasse K ja die Methoden all ihrer allgemeineren (und damit in der Hierarchie weiter oben stehenden) Superklassen, womit gewährleistet ist, dass Objekte von K mindestens alle die Nachrichten verarbeiten können, die Objekte ihrer Superklassen verstehen. Und genau dies – die Frage, welche Nachrichten ein Objekt versteht – war ja unser Kriterium bei der Klassifikation. Und damit *ist* jeder Pudel ein Hund, jeder Kinderbuchladen *ist* ein Buchladen und jeder Buchladen *ist* ein Einzelhandelsgeschäft, ganz wie in der realen Welt.

Wie wir in den folgenden Kapiteln noch sehen werden, ist das Klassifizieren und Spezialisieren eine weitverbreitete Technik, um Wissen und Verfahren zu strukturieren. Die Nutzung dieser Techniken für die Programmierung ist ein zentraler Aspekt der Objektorientierung.

Ad-hoc-Aufgabe 1

Gegeben seien die folgenden beiden Listen von Objekten (bzw. Klassen von Objekten) und Nachrichten:

Klassen: Fahrrad, Lebewesen, Biene, Motorrad, Motorfahrzeug, Objekt, Blume, Auto, Fortbewegungsmittel

Nachrichten: starteMotor, transportierePerson, pflanzeDichFort, sammleHonig, verwelke

1. Überlegen Sie, welche Objekte welche Nachrichten verstehen könnten.
2. Versuchen Sie, die Klassen anhand dieser Eigenschaft in einer Klassenhierarchie analog zu Abbildung 1.1 anzuordnen.
3. Ordnen Sie jede Nachricht der Klasse zu, die in der Hierarchie möglichst weit oben steht und für die gilt, dass alle ihre Objekte (incl. Objekte ihrer Subklassen) diese Nachricht verstehen.
4. Welche der Klassen würden Sie in dieser Hierarchie als abstrakte Klassen ansehen? Was fällt Ihnen bzgl. deren Anordnung auf?
5. Wenn Sie jetzt eine zusätzliche Klasse Reitpferde in Ihre Hierarchie aufnehmen sollten, welche Folgen hätte das für die Hierarchie?

1.1.2 Abgrenzung zur prozeduralen Programmierung

Wie im vorigen Abschnitt erläutert, finden sich einige Aspekte der imperativ-prozeduralen Programmierung auch in der objektorientierten Programmierung wieder. Das betrifft insbesondere die Methoden, welche auf den ersten Blick eine starke Gemeinsamkeit mit Prozeduren aufweisen, und die Attribute, in denen ein Objekt seinen Zustand „aufbewahrt".

Diese Parallelen sind einerseits recht praktisch, weil sie uns erlauben, bestimmte Sachverhalte als aus der prozeduralen Programmierung bekannt vorauszusetzen und darauf aufzubauen. So bedarf es zum Beispiel fast keiner besonderen Erwähnung, dass ein Empfängerobjekt einer Nachricht seinen Zustand ändern kann, indem die nachrichtenverarbeitende Methode eine Zuweisung an eines der Attribute des Objekts vornimmt.

Andererseits bringen diese Gemeinsamkeiten aber das Risiko mit sich, dass Dinge als gleich betrachtet werden, die es letztlich gar nicht sind. Wir wollen daher im Folgenden einige zentrale Eigenschaften der prozeduralen und der objektorientierten Programmierung einander gegenüberstellen, um die Unterschiede zu verdeutlichen. Dabei werden wir bzgl. der objektorientierten Programmierung auch einige weitere Aspekte kurz anreißen, die im weiteren Verlauf des Lehrtextes noch näher erläutert werden.

1.1.2.1 Prozedurale Programmierung

Modellierung von Information und Verarbeitung In der prozeduralen Programmierung ist die Modellierung der Informationen von der Modellierung der Verarbeitung klar getrennt. *Informationen* werden im Wesentlichen durch Grunddaten (ganze Zahlen, boolesche Werte, Zeichen, Zeichenketten usw.) modelliert, die in Variablen gespeichert werden. Variablen lassen sich üblicherweise zu Arrays oder Records (Verbünden) organisieren, um komplexere Datenstrukturen zu realisieren. Das Grundmodell der *Verarbeitung* in prozeduralen Programmen ist die Zustandsänderung. Die Verarbeitung wird modelliert als eine Folge von Zustandsübergängen, wobei sich der globale Zustand aus den Zuständen der Variablen zusammensetzt.

Prozedur

Sicht der Programmierenden Eine *Prozedur* ist eine benannte, parametrisierte Anweisung, die meistens zur Erledigung einer bestimmten Aufgabe bzw. Teilaufgabe dient. Eine Prozedur kann zur Erledigung ihrer Aufgaben andere Prozeduren aufrufen. Für die Programmierenden heißt dies, dass sie geeignete Prozeduren suchen (bzw. nötigenfalls programmieren) und aufrufen, welche auf den – passiven – Daten die von ihnen gewünschten Zustandsänderungen vollziehen.

Strukturierung, Datenkapselung Das Grundmodell der prozeduralen Programmierung erlaubt in natürlicher Weise die Strukturierung der Verarbeitung. Es bietet aber wenig Möglichkeiten, um Teile des Zustands mit den auf ihm operierenden Prozeduren zusammenzufassen und die Kapselung von Daten zu erreichen.

Erweiterbarkeit Prozedurale Programmierung wird meist im Zusammenhang mit strenger Typisierung behandelt. Typkonzepte verbessern zwar die statische Überprüfbarkeit von Programmen, erschweren aber deren Anpassbarkeit. Die Typen der Parameter einer Prozedur `p` sind fest vorgegeben. Wird ein Typ erweitert oder modifiziert, entsteht ein neuer Typ, dessen Elemente von `p` nicht mehr akzeptiert werden, selbst wenn die Änderungen keinen Einfluss auf die Bearbeitung hätten. Damit vorhandene Programmteile auch mit dem neuen Typ arbeiten können, sind meist umfangreiche Änderungen an diesen erforderlich, was eine Neuübersetzung aller betroffenen Teile nötig macht.

Zuordnung Prozeduraufruf - Prozedurausführung Jeder Programmstelle mit einem Prozeduraufruf ist eindeutig eine als Folge dieses Aufrufs auszuführende Prozedur zugeordnet. Sollen unterschiedliche Daten zu unterschiedlichen Programmabläufen führen, müssen entsprechende Fallunterscheidungen explizit programmiert werden. Die Zuordnung der auszuführenden Prozedur zu einem bestimmten Aufruf (das sog. *Binden* des Aufrufs) erfolgt typischerweise bereits bei der Übersetzung des Programms durch einen Compiler. Man spricht hier von *statischem Binden*.

statisches Binden

Parallelität Die prozedurale Programmierung basiert auf einem sequentiellen Ausführungsmodell. Um Parallelität ausdrücken zu können, muss das Grundmodell erweitert werden, beispielsweise indem die parallele Ausführung von Anweisungen oder Prozeduren unterstützt wird oder indem zusätzliche Sprachelemente zur Verfügung gestellt werden (z. B. Prozesse).

1.1.2.2 Objektorientierte Programmierung

Modellierung von Information und Verarbeitung In der objektorientierten Programmierung bilden Objekte eine Einheit aus Daten und den auf ihnen definierten Operationen (in Form von Methoden). Die Gesamtheit der Daten in einem objektorientierten Programm ist auf die einzelnen Objekte verteilt.

Sicht der Programmierenden Es gibt keine global arbeitenden Prozeduren bzw. Operationen. Jede Methode gehört zu einem Objekt und lässt sich von außen nur über das Schicken einer Nachricht an dieses Objekt auslösen. Für die Programmierenden ergibt es daher i. A. keinen Sinn, nach einer globalen Methode zu suchen (oder sie zu schreiben), um diese aufrufen zu können. Wenn etwas getan werden soll, soll es normalerweise an einem bestimmten bekannten Objekt getan werden, dessen Zustand sich dadurch ändern soll[3]. Die Programmierenden sorgen also dafür, dass diesem Objekt eine passende

[3]Einige Programmiersprachen, so auch Java, weichen von diesem Grundkonzept ab und bieten sogenannte *Klassenmethoden* an (siehe auch Abschnitt 2.1.3.2). Klassenmethoden ähneln Prozeduren der prozeduralen Programmierung und können unabhängig von einem Objekt aufgerufen werden. Meist handelt es sich dabei um rein funktionale Methoden, die keine Zustände irgendeines Objekts verändern, sondern lediglich eine bestimmte Aktion auslösen oder aus einem als Methodenparameter übergebenen Eingangswert einen Ausgangswert berechnen.

Nachricht geschickt wird. Nötigenfalls sorgen sie durch Programmierung einer entsprechenden Methode des Objekts dafür, dass es diese Nachricht verarbeiten kann.

Strukturierung, Datenkapselung Jedes Objekt hat eine klar festgelegte Schnittstelle, die beschreibt, welche Nachrichten es „versteht". Diese inhärente Schnittstellenbildung erlaubt zum einen eine Strukturierung durch Klassifizierung, zum anderen unterstützt sie die Kapselung von Daten: Wenn auf den Zustand eines Objekts von anderen Objekten nur über Methoden zugegriffen wird, hat ein Objekt die vollständige Kontrolle über seine Daten[4]. Insbesondere kann es die Konsistenz zwischen Attributen gewährleisten und verbergen, welche Daten es in Attributen hält und welche Daten es erst auf Anforderung berechnet.

Erweiterbarkeit Das Nachrichtenmodell in der objektorientierten Programmierung und die Klassifizierung von Objekten danach, welche Nachrichten sie verstehen, bringt wesentliche Vorteile für eine gute Erweiterbarkeit von Programmen mit sich.

Betrachten wir dazu ein Programm, das Behälter für druckbare Objekte implementiert, d. h. für Objekte verschiedener Klassen, deren Gemeinsamkeit darin besteht, dass sie die Nachricht `print` verstehen, was sich in einer gemeinsamen Superklasse ausdrückt, die z. B. `Printable` heißen könnte. Die Behälterobjekte sollen eine Methode `printAll` besitzen, die jedem Objekt im Behälter die Nachricht `print` schickt, woraufhin abhängig von der Art des Empfängerobjekts unterschiedlicher Programmcode ausgeführt wird.

In einem prozeduralen Programm müssten wir in `printAll` eine Fallunterscheidung vornehmen und je nach Art des druckbaren Objekts unterschiedliche Prozeduren aufrufen. In einem objektorientierten Programm ist diese Fallunterscheidung nicht nötig, weil hier jedes Objekt, welches die Nachricht `print` erhält, immer *seine* spezifische Methode ausführt. Die Unterscheidung der unterschiedlichen Objekte erfolgt also implizit durch den Nachrichtenmechanismus.

Auch bei einer Erweiterung des Programms um neue Klassen druckbarer Objekte mit anderen Eigenschaften ist keine Änderung am Behälter nötig: Man modelliert die neue Klasse als weitere Subklasse der Klasse `Printable` (oder als Subklasse einer der bereits vorhandenen Subklassen von `Printable`) und passt die Implementierung der Methode `print` den neuen Bedürfnissen entsprechend an.

Die neuen Objekte *sind* damit druckbare Objekte und können an allen Stellen eingesetzt werden, an denen die alten Objekte zulässig sind. Auf diesen Aspekt der objektorientierten Programmierung wird im Zusammenhang mit *Subtyping* genauer eingegangen (siehe u. a. Kapitel 3). *Subtyping*

[4]Diese sehr absolute Aussage werden wir im Laufe des Lehrtextes leider ein wenig relativieren müssen.

Die Tatsache, dass es bei einem Nachrichtenversand von der Art des konkreten Nachrichtenempfängers abhängt, welche Methode zur Ausführung gelangt, ist eng verbunden mit dem im folgenden Absatz beschriebenen Mechanismus.

Nachrichtenversand - Methodenausführung Im Gegensatz zur prozeduralen Programmierung ist es bei der objektorientierten Programmierung im Allgemeinen *nicht* möglich, an der Stelle, an der ein Nachrichtenversand erfolgt, vorherzusagen, welche Methode in dessen Folge zur Ausführung gelangen wird. Zwar wird es immer die Methode sein, die genau das Objekt, dem wir die Nachricht senden, für diese Nachricht vorgesehen hat, schließlich ist die Zuständigkeit eines Objekts dafür, wie es auf eine Nachricht reagiert, eines der Grundkonzepte der objektorientierten Programmierung. In der Praxis liegt aber die Schwierigkeit darin, zu entscheiden, welches Objekt überhaupt der Nachrichtenempfänger ist. Z. B. wird im Falle des im vorigen Abschnitt erwähnten Behälters erst zur Laufzeit klar sein, welche druckbaren Objekte sich in ihm befinden. Je nachdem, um welche Objekte es sich handelt, *müssen* aber unterschiedliche Methoden zur Ausführung kommen, wenn ihnen vom Behälterobjekt die Nachricht `print` gesendet wird.

dynamisches Binden

Die Zuordnung der auszuführenden Methode zu einem bestimmten Nachrichtenversand kann bei objektorientierten Programmen also in den meisten Fällen erst zur Laufzeit erfolgen. Man spricht hier von *dynamischem Binden* oder *dynamischer Methodenwahl*. Das dynamische Binden ist letztlich die technische Voraussetzung dafür, dass die bei der prozeduralen Programmierung nötige explizite Fallunterscheidung in der objektorientierten Programmierung dadurch ersetzt werden kann, dass schlicht immer die Methode desjenigen Objekts zum Einsatz kommt, welches der Empfänger der betreffenden Nachricht ist.

statisches Binden

Die hier beschriebene Notwendigkeit, Methodenaufrufe dynamisch zu binden, gilt *nicht* für die bereits in einer Fußnote auf Seite 10 erwähnten Klassenmethoden: Da sie zu keinem Objekt gehören, sondern über den Namen „ihrer“ Klasse aufgerufen werden, steht bereits zur Übersetzungszeit fest, welche Methode als Resultat eines Aufrufs ausgeführt wird, nämlich eben die dem Aufruf entsprechende Methode „dieser“ Klasse. Klassenmethoden werden daher – wie Prozeduren bei der prozeduralen Programmierung – statisch gebunden.

Parallelität Durch die Aufteilung der Verarbeitungsprozesse auf mehrere Objekte ermöglicht das Grundmodell der objektorientierten Programmierung vom Konzept her insbesondere eine natürliche Behandlung von parallelen und verteilten Prozessen der realen Welt. Das Konzept der inhärenten Parallelität wird von den meisten objektorientierten Programmiersprachen allerdings **nicht** unterstützt. Methoden werden standardmäßig sequentiell ausgeführt, auch über Objektgrenzen hinweg. Parallelität muss in diesen Sprachen, ähnlich wie in imperativen Programmiersprachen, explizit durch Sprachkonstrukte eingeführt werden. In Java gibt es dafür das Konzept der Threads, das in Kapitel 13 ausführlich vorgestellt wird.

1.1.3 Objektorientierung als Antwort auf softwaretechnische Anforderungen

Objektorientierte Programmierung ist mittlerweile fast 50 Jahre alt. Bereits die Programmiersprache *Simula 67* besaß alle wesentlichen Eigenschaften für die objektorientierte Programmierung (siehe z. B. [Lam88]). Die Erfolgsgeschichte der objektorientierten Programmierung hat allerdings erst Anfang der achtziger Jahre richtig an Fahrt gewonnen, stark getrieben von der Programmiersprache *Smalltalk* (siehe [GR89]) und der zugehörigen Entwicklungsumgebung. Es dauerte nochmals ein Jahrzehnt, bis die objektorientierte Programmierung auch in der kommerziellen Programmentwicklung nennenswerte Bedeutung bekommen hat. Mittlerweile hat objektorientierte Programmierung eine weite Verbreitung erlangt und viele ursprünglich nicht objektorientierte Programmiersprachen wurden um entsprechende Möglichkeiten erweitert.

Simula

Smalltalk

Es ist schwer im Einzelnen zu klären, warum es so lange gedauert hat, bis die objektorientierten Techniken breitere Beachtung erfahren haben, und warum sie nun so breite Resonanz finden. Sicherlich spielen bei dieser Entwicklung viele Aspekte eine Rolle – das geht beim Marketing los und macht bei ästhetischen Überlegungen nicht halt. Wir beschränken uns hier auf einen inhaltlichen Erklärungsversuch: Objektorientierte Konzepte sind kein Allheilmittel, sie können aber einen wichtigen Beitrag zur Lösung bestimmter *softwaretechnischer Probleme* leisten. In dem Maße, in dem diese Problemklasse in Relation zu anderen softwaretechnischen Problemen an Bedeutung gewonnen hat, haben auch die objektorientierten Techniken an Bedeutung gewonnen und werden voraussichtlich weiter an Bedeutung gewinnen.

Vier softwaretechnische Aufgabenstellungen stehen in einer sehr engen Beziehung zur Entwicklung objektorientierter Techniken und Sprachen:

1. Softwaretechnische Simulation,
2. Konstruktion interaktiver, grafischer Bedienoberflächen,
3. Programm-Wiederverwendung und
4. Verteilte Programmierung.

Nach einer kurzen Erläuterung dieser Bereiche werden wir ihre Gemeinsamkeiten untersuchen.

1. Simulation: Grob gesprochen lassen sich zwei Arten von Simulation unterscheiden: die Simulation *kontinuierlicher* Prozesse, beispielsweise die numerische Berechnung von Klimavorhersagen im Zusammenhang mit dem Treibhauseffekt, und die Simulation *diskreter* Vorgänge, beispielsweise die Simulation des Verkehrsflusses an einer Straßenkreuzung oder die virtuelle Besichtigung eines geplanten Gebäudes auf Basis eines Computermodells. Wir betrachten im Folgenden nur die diskrete Simulation. Softwaretechnisch sind dazu drei Aufgaben zu erledigen:

1. Modellierung der statischen Komponenten des zugrunde liegenden Systems.
2. Beschreibung der möglichen Dynamik des Systems.
3. Test und Analyse von Abläufen des Systems.

Für das Beispiel der Simulation des Verkehrsflusses an einer Straßenkreuzung heißt das: Die Straßenkreuzung mit Fahrspuren, Bürgersteigen, Übergängen usw. muss modelliert werden, Busse, Autos und Fahrräder müssen mit ihren Abmessungen und Bewegungsmöglichkeiten beschrieben werden (Aufgabe 1). Die Dynamik der Objekte dieses Modells muss festgelegt werden, d. h. die möglichen Ampelstellungen, das Erzeugen neuer Fahrzeuge an den Zufahrten zur Kreuzung und die Bewegungsparameter der Fahrzeuge (Aufgabe 2). Schließlich muss eine Umgebung geschaffen werden, mit der unterschiedliche Abläufe auf der Kreuzung gesteuert, getestet und analysiert werden können (Aufgabe 3).

2. Grafische Bedienoberflächen: Interaktive, grafische Bedienoberflächen ermöglichen die nichtsequentielle, interaktive Steuerung von Anwendungsprogrammen über direkt manipulierbare, grafische Bedienelemente wie Schaltflächen, Auswahlmenüs und Eingabefenster. Der Konstruktion grafischer Bedienoberflächen liegen eine ergonomische und zwei softwaretechnische Fragestellungen zugrunde:

1. Wie muss eine Oberfläche gestaltet werden, um der Modellvorstellung der Benutzenden von der gesteuerten Anwendung gerecht zu werden und eine leichte Bedienbarkeit zu ermöglichen?
2. Die Benutzenden möchten quasi-parallel arbeiten, z. B. in einem Fenster eine Eingabe beginnen, dann, bevor sie die Eingabe beenden, eine andere Eingabe berichtigen und eine Information in einem anderen Fenster erfragen, dann ggf. eine Anwendung starten und ohne auf deren Ende zu warten, mit der erstgenannten Eingabe fortfahren. Ein derartiges Verhalten wird von einem sequentiellen Programmiermodell nicht unterstützt: Wie sieht ein gutes Programmiermodell dafür aus?
3. Das Verhalten einer Oberflächenkomponente ergibt sich zum Großteil aus der Standardfunktionalität für die betreffende Komponentenart und nur zum geringen Teil aus Funktionalität, die spezifisch für die Komponente programmiert wurde. Beispielsweise möchte man zu einer Schaltfläche nur programmieren, was bei einem Mausklick getan werden soll. Die Zuordnung von Mausklicks zur Schaltfläche, die Verwaltung und das Weiterreichen von Mausbewegungen und anderen Ereignissen sollte bereits als Standardfunktionalität zur Verfügung stehen. Wie lässt sich diese Standardfunktionalität in Form von Oberflächenbausteinen so zur Verfügung stellen, dass sie programmtechnisch gut, sicher und flexibel handhabbar ist?

3. **Wiederverwendung von Programmteilen** Zwei Hauptprobleme stehen bei der Wiederverwendung von Programmteilen im Mittelpunkt:

   1. Wie finde ich zu einer gegebenen Aufgabenstellung einen Programmbaustein mit Eigenschaften, die den gewünschten möglichst nahe kommen?
   2. Wie müssen Programme strukturiert und parametrisiert sein, um sich für Wiederverwendung zu eignen?

   Wesentliche Voraussetzung zur Lösung des ersten Problems ist die Spezifikation der Eigenschaften der Programmbausteine, insbesondere derjenigen Eigenschaften, die an den Schnittstellen, d. h. für die Benutzenden sichtbar sind. Das zweite Problem rührt im Wesentlichen daher, dass man selten zu einer gegebenen Aufgabenstellung einen fertigen, passenden Programmbaustein findet. Programme müssen deshalb gut anpassbar und leicht erweiterbar sein, um sich für Wiederverwendung zu eignen. Derartige Anpassungen sollten möglich sein, ohne den Programmtext der verwendeten Bausteine manipulieren zu müssen.

4. **Verteilte Programmierung** Die Programmierung verteilter Anwendungen – oft kurz als verteilte Programmierung bezeichnet – soll es ermöglichen, dass Programme, die auf unterschiedlichen Rechnern laufen, miteinander kommunizieren und kooperieren können und dass Daten und Programmteile über digitale Netze automatisch verteilt bzw. beschafft werden können. Demzufolge benötigt die verteilte Programmierung ein Programmiermodell,

   - in dem räumliche Verteilung von Daten und Programmteilen dargestellt werden kann,
   - in dem Parallelität und Kommunikation in natürlicher Weise beschrieben werden können und
   - das eine geeignete Partitionierung von Daten und Programmen in übertragbare Teile unterstützt.

In der Einleitung zu diesem Abschnitt wurde der Erfolg objektorientierter Techniken teilweise damit erklärt, dass sie sich besser als andere Techniken eignen, um die skizzierten softwaretechnischen Aufgabenstellungen zu bewältigen. Schrittweise wollen wir im Folgenden untersuchen, woher die bessere Eignung für diese Aufgaben kommt. Dazu stellen wir zunächst einmal gemeinsame Anforderungen zusammen. Diese Anforderungen dienen uns in den kommenden Abschnitten als Grundlage, um die spezifischen Aspekte des objektorientierten Programmiermodells herauszuarbeiten.

Frage. Welche Anforderungen treten in mehreren der skizzierten softwaretechnischen Aufgabenstellungen auf?

Die Aufgabenstellungen sind zwar in vielen Aspekten sehr unterschiedlich, aber jede von ihnen stellt zumindest zwei der folgenden drei konzeptionellen Anforderungen:

1. Sie legt ein inhärent paralleles Ausführungsmodell nahe, mit dem insbesondere Bezüge zur realen Welt modelliert werden können. („Inhärent parallel“ bedeutet, dass Parallelität eine Eigenschaft des grundlegenden Ausführungsmodells ist und nicht erst nachträglich hinzugefügt werden muss.)

2. Die Strukturierung der Programme in kooperierende Programmteile mit klar definierten Schnittstellen spielt eine zentrale Rolle.

3. Anpassbarkeit, Klassifikation und Spezialisierung von Programmteilen sind wichtige Eigenschaften und sollten möglich sein, ohne bestehende Programmtexte manipulieren zu müssen.

In der Simulation ermöglicht ein paralleles Ausführungsmodell eine größere Nähe zwischen dem simulierten Teil der realen Welt und dem simulierenden Softwaresystem. Dabei sollten Programme so strukturiert sein, dass diejenigen Daten und Aktionen, die zu einem Objekt der realen Welt gehören, innerhalb des Programms zu einer Einheit zusammengefasst sind. Darüber hinaus sind Klassifikationshierarchien bei der Modellierung sehr hilfreich: Im obigen Simulationsbeispiel ließen sich dann die Eigenschaften aller Fahrzeuge gemeinsam beschreiben. Die Eigenschaften speziellerer Fahrzeugtypen (Autos, Busse, Fahrräder) könnte man dann durch Verfeinerung der Fahrzeugeigenschaften beschreiben.

Wie bereits skizziert, liegt auch interaktiven, grafischen Bedienoberflächen ein paralleles Ausführungsmodell zugrunde. Der Bezug zur realen Welt ergibt sich hier aus der Interaktion mit den Benutzenden. Anpassbarkeit, Klassifikation und die Möglichkeit der Spezialisierung von Programmteilen sind bei Bedienoberflächenbaukästen besonders wichtig. Sie müssen eine komplexe und mächtige Standardfunktionalität bieten, um den Programmierenden Arbeit zu sparen. Sie können andererseits aber nur unfertige Oberflächenkomponenten bereitstellen, die erst durch Spezialisierung ihre dedizierte, für den speziellen Anwendungsfall benötigte Funktionalität erhalten.

Die unterschiedlichen Formen der Wiederverwendung von Programmen werden wir in späteren Kapiteln näher analysieren. Im Allgemeinen stehen bei der Wiederverwendung die Anforderungen 2 und 3 im Vordergrund. Wenn man den Begriff „Wiederverwendung“ weiter fasst und z. B. auch dynamisches Laden von Programmkomponenten über eine Netzinfrastruktur oder sogar das Nutzen im Netz verfügbarer Dienste als Wiederverwendung begreift, spielen auch Aspekte der verteilten Programmierung eine wichtige Rolle für die Wiederverwendung.

Bei der verteilten Programmierung ist ein paralleles Ausführungsmodell sinnvoll, dessen Bezugspunkte in der realen Welt sich durch die räumliche Verteilung der kooperierenden Programmteile ergeben. Darüber hinaus bildet Anforderung 2 eine Grundvoraussetzung für die verteilte Programmierung: Es muss klar definiert sein, wer kooperieren kann und wie die Kommunikation im Einzelnen aussieht.

Fazit. Die Entwicklung objektorientierter Konzepte und Sprachen war und ist eng verknüpft mit der Erforschung recht unterschiedlicher softwaretechnischer Aufgabenstellungen (das ist eine Beobachtung). Aus diesen Aufgabenstellungen resultieren bestimmte Anforderungen an die Programmierung (das ist ein Analyseergebnis). Die Konzepte und Techniken der objektorientierten Programmierung sind im Allgemeinen besser als andere Programmierparadigmen geeignet, diese Anforderungen zu bewältigen (dies ist – immer noch – eine Behauptung). Ein Ziel dieses Lehrtextes ist es, diese Behauptung argumentativ zu untermauern und dabei zu zeigen, wie die bessere Eignung erreicht werden soll und dass dafür auch ein gewisser Preis zu zahlen ist.

1.2 Programmiersprachlicher Hintergrund

Dieser Abschnitt stellt den programmiersprachlichen Hintergrund bereit, auf den wir uns bei der Behandlung objektorientierter Sprachkonzepte in den folgenden Kapiteln stützen werden. Er gliedert sich in drei Teile:

1. Zusammenfassung grundlegender Sprachkonzepte von imperativen und objektorientierten Sprachen am Beispiel von Java.
2. Objektorientierte Programmierung mit Java.
3. Überblick über existierende objektorientierte Sprachen.

Der erste Teil bietet darüber hinaus eine Einführung in die Basisdatentypen, Kontrollstrukturen und deren Syntax in Java.

1.2.1 Grundlegende Sprachmittel am Beispiel von Java

Bei den meisten objektorientierten Programmiersprachen werden objektlokale Berechnungen mit imperativen Sprachmitteln beschrieben. Im Folgenden sollen die in den späteren Kapiteln benötigten Sprachmittel systematisch zusammengefasst werden, um eine begriffliche und programmtechnische Grundlage zu schaffen. Begleitend werden wir zeigen, wie diese Sprachmittel in Java umgesetzt sind. Dabei werden wir uns auf eine knapp gehaltene Einführung beschränken, die aber alle wesentlichen Aspekte anspricht. Eine detaillierte Darstellung findet sich in der Java-Sprachspezifikation [⇝JavaLangSpec].

1.2.1.1 Objekte und Werte: Eine begriffliche Abgrenzung

Unser bisheriges Bild von Objekten sieht unter Anderem vor, dass sie eine Identität und einen Zustand haben, wobei zwei Objekte gleicher Art, die den gleichen Zustand haben, dennoch nicht dasselbe Objekt sind und eine Änderung des Zustands eines Objekts nichts an dessen Identität ändert.

Für viele Objekte erscheint diese Sichtweise sofort einleuchtend und entspricht unseren Erfahrungen aus der realen Welt: Ein rotes Auto eines bestimmten Modells und Baujahrs ist nicht dasselbe Objekt wie ein anderes rotes Auto dieses Modells und Baujahrs. Und wenn wir ein Auto anlassen, ändert dieses zwar seinen Zustand, es bleibt aber dasselbe Auto.

Es gibt aber auch Objekte, für welche das beschriebene Bild nicht so recht passend erscheint. Nehmen wir als Beispiel die Zahl Zwei. Nach dem bisher Gesagten könnte es sich um ein Objekt einer Klasse Ganzzahl handeln, welchem man z. B. die Nachricht `plus` senden kann. Dieser Nachricht könnte man eine Referenz auf eine andere Ganzzahl, etwa eine Drei, als Parameter mitgeben. Allerdings wirft diese Vorgehensweise die Frage auf, was die Zwei mit der Nachricht anfängt. Wenn sie sich irgendwie in eine Fünf verwandeln würde, wäre sie keine Zwei mehr! Auch eine Zustandsänderung der Zwei ergibt keinen Sinn, denn wenn die Zwei überhaupt so etwas wie einen Zustand hat, dann besteht dieser ja eben darin, die Zwei zu sein.

Begrifflich unterscheiden wir deshalb zwischen *Objekten* und *Werten* (engl. *objects* und *values*). Prototypisch für Objekte sind materielle Gegenstände (Autos, Lebewesen etc.). Prototypische Werte sind Zahlen, Buchstaben, Mengen und (mathematische) Funktionen. Die Begriffe „Objekt“ und „Wert“ sind fundamentaler Natur und lassen sich nicht mittels anderer Begriffe definieren. Wir werden deshalb versuchen, sie durch charakteristische Eigenschaften voneinander abzugrenzen:

Objekt vs. Wert

1. Zustand: Objekte haben einen veränderbaren Zustand (ein Auto kann eine Beule bekommen, ein Mensch eine neue Frisur; eine Mülltonne kann geleert werden). Werte sind abstrakt und können nicht verändert werden (es macht keinen Sinn, die Zahl Zwei verändern zu wollen; entnehme ich einer Menge ein Element, erhalte ich eine andere Menge).

2. Identität: Objekte besitzen eine Identität, die vom Zustand unabhängig ist. Objekte können sich also völlig gleichen, ohne identisch zu sein (man denke etwa an zwei baugleiche Autos). Insbesondere kann man durch Klonen/Kopieren eines Objekts `obj1` ein anderes Objekt `obj2` erzeugen, das `obj1` in allen Eigenschaften gleicht, aber nicht mit ihm identisch ist. Zukünftige Änderungen des Zustands von `obj1` haben dann keinen Einfluss auf den Zustand von `obj2` und umgekehrt.

3. Lebensdauer: Objekte besitzen eine Lebensdauer, insbesondere gibt es Operationen, um Objekte zu erzeugen, ggf. auch um sie zu löschen. Werte besitzen keine beschränkte Lebensdauer, sondern existieren quasi ewig.

4. Aufenthaltsort: Objekten kann man üblicherweise einen Aufenthaltsort, beschrieben durch eine Adresse, zuordnen. Werte lassen sich nicht lokalisieren.

5. Verhalten: Objekte stellt man sich als aktiv vor, d. h. sie reagieren auf Nachrichten und weisen dabei ein zustandsabhängiges Verhalten auf. Werte besitzen kein „Eigenleben“: Auf ihnen operieren Funktionen, die Eingabewerte zu Ergebniswerten in Beziehung setzen.

Der konzeptionell relativ klare Unterschied zwischen Objekten und Werten wird bei vielen programmiersprachlichen Realisierungen nur zum Teil beibehalten. Zur Vereinheitlichung behandelt man Werte häufig wie Objekte. So werden z. B. in Smalltalk ganze Zahlen als Objekte modelliert. Solche Zahl-Objekte besitzen einen unveränderlichen Zustand, der dem Wert der Zahl entspricht, eine Lebensdauer bis zum Ende der Programmlaufzeit und Methoden, die den arithmetischen Operationen entsprechen. Zahl-Objekte werden als identisch betrachtet, wenn sie denselben Wert repräsentieren. In Java wird eine ähnliche Konstruktion verwendet, um zahlwertige Datentypen als Subtypen des besonderen Typs `Object` behandeln zu können (vgl. Unterabschn. 3.4).

1.2.1.2 Werte, Typen und Variablen in Java

Basis-datentypen

Ein Datentyp beschreibt eine Menge von Werten zusammen mit den darauf definierten Operationen. Java stellt die vordefinierten Basisdatentypen (engl. primitive data types) `byte`, `short`, `int`, `long`, `float`, `double`, `char` und `boolean` zur Verfügung. Wertebereich und der jeweilige Speicherbedarf der Basisdatentypen sind in der folgenden Tabelle zusammengestellt:

`byte`	-128 bis 127	1 Byte
`short`	-32768 bis 32767	2 Byte
`int`	-2147483648 bis 2147483647	4 Byte
`long`	-9223372036854775808L bis 9223372036854775807L	8 Byte
`float`	im Bereich ±3.402823E+38F jeweils 6-7 signifikante Stellen	4 Byte
`double`	im Bereich ±1.797693E+308 jeweils 15 signifikante Stellen	8 Byte
`char`	65536 Unicode-Zeichen[5] Notationsbeispiele: `'a'` `'+'` `'n'` `','` `'u0022'`	2 Byte
`boolean`	`true`, `false`	nicht spezifiziert

Konstante

Die Tabelle zeigt auch, wie die *Konstanten*[6] der Basisdatentypen in Java geschrieben werden. Die Konstanten der Typen `byte`, `short` und `int` werden notationell nicht unterschieden. Die Konstanten des Typs `long` haben ein „`L`“ als Postfix. Bei Gleitkommazahlen kann die Exponentenangabe entfallen. Unicode-Zeichen lassen sich grundsätzlich durch `'\u`*xxxx*`'` in Java ausdrücken, wobei *x* für eine Hexadezimalziffer steht. Ein ASCII-Zeichen *a* lässt sich aber auch direkt als `'a'` notieren. Darüber hinaus bezeichnet `'\t'` das Tabulatorzeichen, `'\n'` das Neue-Zeile-Zeichen, `'\''` das Hochkomma, `'\"'` das Anführungszeichen und `'\\'` den Backslash. Die beiden Konstanten des Typs `boolean` werden durch die Schlüsselwörter `true` und `false` repräsentiert.

Werte in Java

Ein *Wert* in Java ist entweder

- ein Element eines der Basisdatentypen,
- eine Referenz auf ein Objekt (Referenzen auf Variablen gibt es in Java nicht)
- die sog. „leere Referenz“ `null` , die auf kein Objekt verweist (womit sie genau genommen gar keine Referenz ist).

[5] Seit der Veröffentlichung von Unicode 2.0 ist es möglich, mehr als 65536 Unicode-Zeichen zu definieren. Dazu verwendet Java nötigenfalls zwei aufeinanderfolgende char-Zeichen in UTF-16 Kodierung. Sollte Sie das Thema „Unicode in Java“ interessieren, finden Sie hier weitere Informationen: [⇝UnicodeJava]

[6] Der Begriff „Konstante“ ist hier im Sinne von „konstanter Wert“ zu verstehen: Eine als Zahl geschriebene Zwei ist konstant, weil ihr Wert eben auf 2 festgelegt ist. Man spricht hier auch von „unbenannten Konstanten“ oder „literalen Konstanten“.

Da Variablen Speicher für *Werte* sind, ergibt sich aus dem soeben Gesagten insbesondere, dass eine Variable in Java niemals ein Objekt enthalten kann, sondern höchstens eine Referenz auf ein solches. Abbildung 1.2 zeigt das Objekt *obj* und die Variablen `a`, `b`, `i` und `flag`. Wie in der Abbildung zu sehen, stellen wir Objekte als Rechtecke mit runden Ecken und Variablen als Rechtecke mit rechtwinkligen Ecken dar. Die Variablen `a` und `b` enthalten jeweils eine *Referenz* auf *obj*, die grafisch durch einen Pfeil dargestellt wird. Während wir Objektreferenzen durch Pfeile repräsentieren, benutzen wir bei anderen Werten die übliche Darstellung. So enthält die int-Variable `i` den Wert 1998 und die boolesche Variable `flag` den Wert `true`.

Objektreferenz

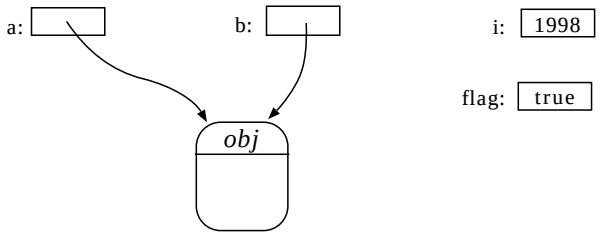

Abbildung 1.2: Referenziertes Objekt und Variablen

Am besten stellt man sich eine Referenz als eine Adresse für ein Objekt vor. Konzeptionell spielt es dabei keine Rolle, ob wir damit eine abstrakte Programmadresse, eine Speicheradresse auf dem lokalen Rechner oder eine Adresse auf einem entfernten Rechner meinen. Lokale Referenzen sind das gleiche wie *Zeiger* in der imperativen Programmierung.

Zeiger

Jeder Wert in Java hat einen *Typ*. Beispielsweise hat der durch die Konstante `true` bezeichnete Wert den Typ `boolean`; die Konstanten `'c'` und `'\n'` bezeichnen Werte vom Typ `char`. Außer den vordefinierten Basisdatentypen gibt es in Java Typen für Objekte. Die Typisierung von Objekten behandeln wir in Kapitel 3 genauer. Für die hier zusammengestellten Grundlagen ist es nur wichtig, dass jedes Objekt in Java einen Typ hat. Da in Java vom Programmkontext her immer klar ist, wann ein Objekt und wann eine Objektreferenz gemeint ist (und man Objekte immer nur über eine entsprechende Referenz ansprechen kann), ist es in Java nicht üblich – wie das etwa in C++ der Fall ist – zwischen dem Typ eines Objekts *obj* und dem Typ der Referenz auf *obj* zu unterscheiden.

Typen in Java

Deshalb werden auch wir zur Vereinfachung der Sprechweise später nicht mehr zwischen Objekten und Objektreferenzen unterscheiden. In diesem Kapitel werden wir weiterhin präzise formulieren; in Lektion 2 werden wir auf den Unterschied durch einen Klammerzusatz aufmerksam machen, wo dies ohne Umstände möglich ist.

Variablen sind Speicher für Werte. In Java sind Variablen (und andere Ausdrücke) typisiert. Ein Typ bezeichnet eine Menge von Werten (Beispiel: „alle Ganzzahlen“) oder Objekten (Beispiel: „alle Personen“) und legt fest, welche Operationen auf diesen Werten möglich sind bzw. welche Nachrichten diese Objekte verstehen. Die Typisierung von Ausdrücken erlaubt es, bereits bei

Variablen

der Kompilierung weitestgehend sicherzustellen, dass auf Werten nur Operationen erfolgen, die für diese Werte auch erlaubt sind und dass einem Objekt nur solche Nachrichten geschickt werden, die es auch versteht [7]. Eine *Variablendeklaration* legt den Typ und Namen der Variablen fest. Folgendes Codefragment deklariert die Variablen `i`, `flag`, `a`, `b`, `s1` und `s2`:

```
int i;
boolean flag;
Object a, b;
String s1, s2;
```

Die Variable `i` kann Zahlen vom Typ `int` speichern, `flag` kann boolesche Werte speichern, `a` und `b` können Referenzen auf je ein Objekt vom Typ `Object` und `s1` und `s2` können Referenzen auf je ein Objekt vom Typ `String` speichern. Wie wir in Kap. 2 sehen werden, sind `Object` und `String` vordefinierte Objekttypen aus der Java-Standardklassenbibliothek.

1.2.1.3 Arrays in Java

Arrays (dt. *Felder*[8]) sind in Java Objekte. Dementsprechend werden sie dynamisch, d. h. zur Laufzeit, erzeugt und Referenzen auf Arrays können an Variablen zugewiesen und als Parameter an Methoden übergeben werden. Ist T ein beliebiger Typ in Java, dann bezeichnet T`[]` den Typ der Arrays mit *Elementtyp* T. Den Operator `[]` nennt man einen *Typkonstruktor*, da er zu einem Elementtyp den entsprechenden Array-Typ konstruiert.

Elementtyp

Typkonstruktor

Jedes Array(-Objekt) besitzt ein unveränderliches Attribut `length` vom Typ `int` und eine bestimmte Anzahl von Attributen vom Elementtyp. Die Attribute vom Elementtyp nennen wir im Folgenden die *Elemente* des Arrays. Die Anzahl der Elemente wird bei der *Erzeugung* des Arrays festgelegt und im Attribut `length` gespeichert. Arrays sind in Java also immer eindimensional. Allerdings können die Elemente eines Arrays andere Arrays referenzieren, sodass mehrdimensionale Arrays als Arrays von Arrays realisiert werden können.

Array-Element

Arrays als Objekte zu realisieren hat den Vorteil, dass sich Arrays auf diese Weise besser in objektorientierte Klassenhierarchien einbetten lassen (Genaueres dazu in Abschnitt 3.5). Außerdem gewinnt man an Flexibilität, wenn zwei- bzw. mehrdimensionale Arrays realisiert werden sollen. Beispielsweise müssen bei zweidimensionalen Arrays in der Programmiersprache Pascal alle Spalten bzw. alle Zeilen die gleiche Länge haben. Bei einer Realisierung mittels Referenzen können die von einem Array referenzierten Arrays auch verschiedene Längen haben; man spricht dann von *Jagged Arrays*. Auch können mehrfach auftretende „Spalten“ bzw. „Zeilen“ durch ein mehrfach referenziertes Array realisiert werden (vgl. Abbildung 1.3).

[7] Hier sei angemerkt, dass typisierte Ausdrücke und entsprechende Prüfungen bei der Kompilierung keineswegs zwingende Eigenschaften objektorientierter Programmiersprachen sind. Smalltalk und Javascript sind wichtige Beispiele für objektorientierte Sprachen, die auf ein solches „Typsystem“ verzichten

[8] In der engl. Literatur wird das Wort „field“ für die in einer Klasse deklarierten Attribute verwendet. Um Verwechslungen zu vermeiden, verwenden wir durchgehend den Begriff „Array“ statt „Feld“.

Folgendes Codefragment deklariert Variablen für zwei eindimensionale Arrays `vor` und `fliegen` und ein zweidimensionales Array `satz`:

```
char[] vor;
char[] fliegen;
char[][] satz;
```

Obige Variablendeklarationen legen dabei lediglich fest, dass die Variablen `vor` und `fliegen` jeweils eine Referenz auf ein Array (beliebiger Länge) enthalten können, dessen Elemente vom Typ `char` sind und dass `satz` eine Referenz auf ein Array mit Elementtyp `char[]` enthalten kann, d. h. die Array-Elemente können (Referenzen auf) char-Arrays speichern. Die Anzahl der Elemente wird in diesen Deklarationen noch nicht festgelegt. Wie eine solche Festlegung erfolgt, wird im folgenden Abschnitt erläutert.

1.2.1.4 Operation, Zuweisungen und Auswertung in Java

Java stellt drei Arten von Operationen zur Verfügung:

1. Operationen, die auf allen Typen definiert sind. Dazu gehören der Test auf Identität == und dessen Umkehrung, der Test auf Nicht-Identität `!=` mit booleschem Ergebnis sowie die *Zuweisungsoperation* = . *Zuweisung*

2. Operationen, die nur für Objektreferenzen bzw. Objekte definiert sind (Methodenaufruf, Objekterzeugung, Typtest von Objekten); diese werden wir in Kap. 2 behandeln.

3. Operationen zum Rechnen mit den Werten der Basisdatentypen; diese sind in Abb. 1.4 zusammengefasst.

Zuweisungen In Java ist die Zuweisung (wie in C) eine Operation mit Seiteneffekt: Sie weist der Variablen auf der linken Seite des Zuweisungszeichens „=“ den Wert des Ausdrucks der rechten Seite zu. Die Zuweisung ist aber selbst ebenfalls ein Ausdruck und liefert als Wert das Ergebnis der Zuweisung zurück. Dazu betrachten wir einige Beispiele aufbauend auf den Variablendeklarationen von S. 22:

```
(01)  i = 4;
(02)  flag = (5 != 3);
(03)  flag = (5 != (i = 3));
(04)  a = (b = null);
(05)  char[] vor = new char[3];
(06)  vor[0] = 'v';
(07)  vor[1] = 'o';
(08)  vor[2] = 'r';
(09)  char[] fliegen = {'F','l','i','e','g','e','n'};
(10)  char[][] satz = {fliegen, {'f','l','i','e','g','e','n'}, vor, fliegen};
```

Nach Ausführung von Zeile (01) hat die Variable `i` den Wert `4`. Die Zuweisung `i = 4` selbst liefert den Wert `4`, dieser wird aber nicht weiter verwendet. Nach Ausführung von Zeile (02) hat die Variable `flag` den Wert `true`. Die Zuweisung `flag = (5 != 3)` selbst liefert den Wert `true`, dieser wird

aber nicht weiter verwendet. In Zeile (03) erfolgt zunächst durch `i = 3` eine Zuweisung an die Variable `i`. Diese hat anschließend den Wert `3` und das ist auch das, was die Zuweisung zurückliefert. Diesmal wird der Wert aber weiterverwendet: Er wird auf Ungleichheit mit 5 getestet und das Ergebnis – der Wert `true` – wird der Variablen `flag` zugewiesen. Nach Zeile (03) hat also die Variable `i` den Wert `3` und die Variable `flag` hat den Wert `true`. Die gesamte Anweisung `flag = (5 != (i=3))` liefert den Wert `true`, der wieder nicht weiter verwendet wird. In Zeile (04) erfolgt zunächst durch `b = null` eine Zuweisung an die Variable `b`. Diese hat anschließend den Wert `null` und das ist auch das, was die Zuweisung zurückliefert. Dieser Wert wird dann der Variablen `a` zugewiesen. Nach Zeile (04) haben also die Variablen `a` und `b` den Wert `null`. Die gesamte Anweisung `a = (b = null)` liefert den Wert `null`, der wieder nicht weiter verwendet wird.

Die Zeilen (05) - (10) zeigen im Zusammenhang mit Arrays relevante Operationen (auf die Betrachtung der Zuweisungen in ihrer Rolle als Ausdrücke verzichten wir dabei). In Zeile (05) wird die Variable `vor` deklariert und mit einem Array der Länge 3 initialisiert [9]. In den Zeilen (06) - (08) werden die drei Array-Elemente initialisiert. Man beachte, dass die Elemente von 0 bis $length - 1$ indiziert werden. In Zeile (09) wird die Variable `fliegen` deklariert und mit einem Array der Länge 7 initialisiert. Die Länge berechnet der Compiler dabei aus der Länge der angegebenen Aufzählung von Ausdrücken (in dem Beispiel ist jeder Ausdruck eine char-Konstante). Diese Form der Initialisierung eines Arrays durch Aufzählung seiner Elemente in geschweiften Klammern ist nur im Kontext der Deklaration einer Variablen gestattet, welcher das neu erzeugte Array direkt zugewiesen wird. In Zeile (10) wird die Variable `satz` deklariert und mit einem vierelementigen Array initialisiert,

- dessen erstes und viertes Element mit dem in Zeile (09) erzeugten Array initialisiert wird,
- dessen zweites Element mit einem neu erzeugten Array initialisiert wird
- und dessen drittes Element mit dem von `vor` referenzierten Array initialisiert wird.

Das resultierende *Objektgeflecht* ist in Abb. 1.3 dargestellt.

Operationen auf Basisdatentypen Die wichtigsten Operationen auf Basisdatentypen sind in Abbildung 1.4 zusammengefasst. Operationen, die auf der Bitdarstellung von Zahlen arbeiten, sowie bestimmte Formen der Zuweisung und Operatoren zum Inkrementieren und Dekrementieren haben wir der Kürze halber weggelassen. Der +-Operator im Zusammenhang mit Objekten des Typs `String` wird auf Seite 29 erläutert.

[9] Präziser: Die Variable wird – da Arrays Objekte sind – mit einer *Referenz* auf dieses Array initialisiert. Dies gilt entsprechend auch für die anderen hier gezeigten Zuweisungen von Arrays an Variablen.

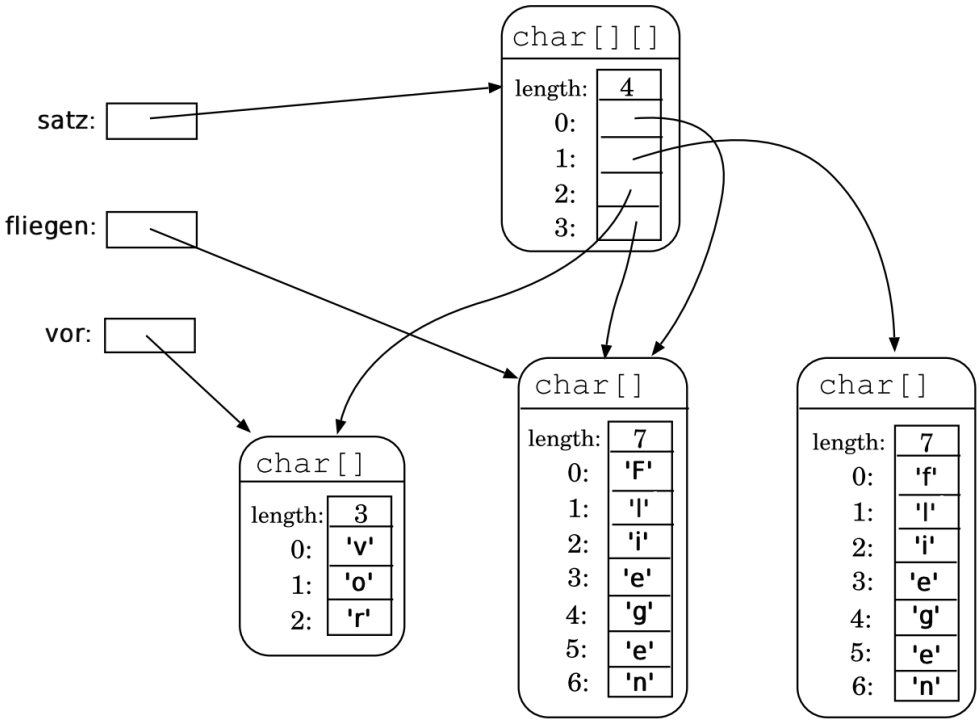

Abbildung 1.3: Die Array-Objekte zum Beispiel

Ausdrücke und deren Auswertung Ein *Ausdruck* (engl. *expression*) ist eine Variable, eine Konstante oder eine Operation angewendet auf Ausdrücke. Wie üblich werden Klammern verwendet, um die Reihenfolge der anzuwendenden Operationen eindeutig zum Ausdruck zu bringen. Außerdem werden, um Klammern einzusparen, gewisse Vorrangregeln beachtet. Z. B. hat $\star$ Vorrang vor + („Punktrechnung vor Strichrechnung“) und alle arithmetischen und logischen Operatoren haben Vorrang vor dem Zuweisungsoperator =. Jeder Ausdruck in Java besitzt einen Typ, der sich bei Variablen aus deren Deklaration ablesen lässt, der sich bei Konstanten aus der Notation ergibt und der bei Operationen deren Ergebnistyp entspricht. Grundsätzlich gilt, dass die Typen der Operanden genau den Argumenttypen der Operationen entsprechen müssen. Um derartige Typgleichheit erzielen zu können, bietet Java die Möglichkeit, Werte eines Typs in Werte eines anderen Typs zu konvertieren. Häufig spricht man anstatt von *Typkonvertierung* auch von Typecasts oder einfach nur von Casts (vom engl. to cast). Ein Ausdruck bildet zusammen mit einem Cast einen neuen Ausdruck, dessen Typ derjenige Typ ist, zu dem gecastet wurde.

Ausdruck

Typkonvertierung (cast)

In Java werden Typkonvertierungen dadurch notiert, dass man den Namen des gewünschten Ergebnistyps, in Klammern gesetzt, dem Wert bzw. Ausdruck voran stellt. Beispielsweise bezeichnet `(long)3` den Wert drei vom Typ `long`, ist also gleichbedeutend mit `3L`. Java unterstützt insbesondere die Typkonvertierung zwischen allen Zahltypen, wobei der Typ `char` als Zahltyp mit Wertebereich 0 bis 65535 betrachtet wird. Vergrößert sich bei der Typkonvertierung der Wertebereich, z. B. von `short` nach `long`, bleibt der Zahlwert unverändert. Andernfalls, z. B. von `int` nach `byte`, führt die Typ-

Operator	Argumenttypen	Ergebnistyp	Beschreibung
`+, -, *, /, %`	`int` × `int`	`int`	ganzzahlige Addition etc.
`+, -, *, /, %`	`long` × `long`	`long`	ganzzahlige Addition etc., wobei `%` den Rest bei ganzzahliger Division liefert
`+, -, *, /`	`float` × `float`	`float`	Gleitkomma-Addition etc.
`+, -, *, /`	`double` × `double`	`double`	Gleitkomma-Addition etc.
`-`	*zahltyp*	*zahltyp*	arithmetische Negation
`<, <=, >, >=`	*zahltyp* × *zahltyp*	`boolean`	kleiner, kleiner-gleich etc., wobei *zahltyp* für `int`, `long`, `float` oder `double` steht
`!`	`boolean`	`boolean`	logisches Komplement
`&, |, ^`	`boolean`×`boolean`	`boolean`	logische Operationen Und, Oder und ausschließendes Oder (xor)
`&&, ||`	`boolean`×`boolean`	`boolean`	nicht strikte Und-/Oder-Operation, d. h. rechter Operand wird ggf. nicht ausgewertet
`_?_:_`	`boolean` × *typ* × *typ*	*typ*	bedingter Ausdruck (Bedeutung weiter unten erläutert)

Abbildung 1.4: Operationen der Basisdatentypen

konvertierung im Allgemeinen zu einer Verstümmelung des Wertes. Um die Lesbarkeit der Programme zu erhöhen, werden bestimmte Konvertierungen in Java implizit vollzogen: z. B. werden alle ganzzahligen Typen, wo nötig, in ganzzahlige Typen mit größerem Wertebereich konvertiert und ganzzahlige Typen werden, wo nötig, in Gleitkommatypen konvertiert. Die folgenden beiden Beispiele demonstrieren diese implizite Typkonvertierung, links jeweils der Java-Ausdruck, bei dem implizit konvertiert wird, rechts ist die Konvertierung explizit angegeben:

```
585888 * 3L                ((long) 585888) * 3L
3.6 + 45L                  3.6 + ((double) 45L)
```

Man beachte, dass auch implizite Konvertierungen zur Verstümmelung der Werte führen können (beispielsweise bei der Konvertierung von großen `long`-Werten nach `float`).

Auswertung von Ausdrücken

Die *Auswertung* (engl. *evaluation*) eines Ausdrucks ist über dessen Aufbau definiert. Ist der Ausdruck eine Konstante, liefert die Auswertung den Wert der Konstanten. Ist der Ausdruck eine Variable, liefert die Auswertung den Wert, der in der Variablen gespeichert ist. Besteht der Ausdruck aus einer Operation angewendet auf Unterausdrücke, gilt grundsätzlich, dass zuerst die Unterausdrücke von links nach rechts ausgewertet werden und dann die Operation auf die Ergebnisse angewandt wird (*strikte* Auswertung). Abweichend davon wird bei den booleschen Operationen `&&` bzw. `||` der rechte Operand nicht mehr ausgewertet, wenn die Auswertung des linken Operanden `false` bzw. `true` ergibt, da in diesen Fällen das Ergebnis des gesamten Ausdrucks bereits feststeht (*nicht strikte* Auswertung). Entsprechendes gilt für

den bedingten Ausdruck `B ? A1 : A2`. Zu dessen Auswertung werte zunächst den booleschen Ausdruck `B` aus. Wenn er `true` ergibt, werte `A1` aus und liefere dessen Ergebnis, ansonsten `A2`.

normale und abrupte Terminierung der Auswertung

Die Auswertung eines Ausdrucks kann in Java auf zwei Arten terminieren: *normal* mit dem üblichen Ergebnis oder *abrupt*, wenn bei der Ausführung einer Operation innerhalb des Ausdrucks ein Fehler auftritt – z. B. Division durch 0. Im Fehlerfall wird die Ausführung des gesamten umfassenden Ausdrucks sofort beendet und eine Referenz auf ein spezielles Objekt zurückgeliefert, das Informationen über den Fehler bereitstellt (vgl. Abschnitt 1.2.1.8.

Ad-hoc-Aufgabe 2

1. Wie sieht das Objektgeflecht aus, das durch folgende Anweisung erzeugt wird?

```
boolean[][] b = new boolean[2][];
```

2. Überlegen Sie, was passieren wird, wenn anschließend diese Anweisung ausgeführt wird:

```
b[0][0] = true;
```

3. Seien `value` und `newValue` Variablen vom Typ `int`. Welchen Wert hat `value` nach der folgenden Anweisung?

```
value = newValue > value ? newValue : value;
```

1.2.1.5 Ausführung eines Java-Programms

Damit Sie im weiteren Verlauf des Lehrtextes Codebeispiele ausprobieren und mit den vorgestellten Sprachmitteln eigene Versuche anstellen können, sollten Sie – sofern Sie das noch nicht getan haben – spätestens jetzt die Voraussetzungen dafür schaffen, indem Sie das Java-Development-Kit und die Entwicklungsumgebung Eclipse installieren. Die Hinweise dazu haben wir in einen „Vorkurs“ ausgelagert, um sie besser aktuell halten zu können. Sie finden den Vorkurs in der Moodle-Umgebung dieses Moduls.

Die Vorstellung, dass ein objektorientiertes Programm aus einer Menge von Objekten besteht, die einander Nachrichten senden, lässt eine Frage unbeantwortet, nämlich die, wie diese Objekte überhaupt entstehen sollen. Zwar kann ein Objekt als Reaktion auf eine Nachricht weitere Objekte erzeugen, aber zu Beginn der Programmausführung existieren ja noch *keine* Objekte. Für Java lautet die Antwort, dass es neben den uns bereits bekannten Methoden, welche die Fähigkeit von Objekten modellieren, Nachrichten zu verstehen, noch weitere Methoden gibt, welche nicht Objekten zugeordnet sind, sondern Klassen, die bereits in der Fußnote auf Seite 10 erwähnten Klassenmethoden.

Klassenmethoden

Aus einer vereinfachten Sicht von Programmierenden besteht ein Java-

Programm aus einer oder mehreren selbst entwickelten Klassen[10]. Um ein Java-Programm ausführen zu können, muss der Quellcode dieser Klassen zu-
Bytecode nächst mit einem *Compiler* in einen *Zwischencode* (Bytecode) übersetzt werden. Nachdem der Bytecode erzeugt wurde, kann er von einer sogenannten *virtuellen Maschine* ausgeführt werden.

virtuelle Maschine Eine virtuelle Maschine ist selbst ein Programm[11], dem man bei seinem
main-Methode Start als Aufrufparameter die Information übergibt, welche Klasse als Programmeinstiegsklasse dienen soll. Diese Programmeinstiegsklasse muss eine ganz bestimmte Klassenmethode besitzen, die sogenannte *main-Methode*. Wird nun die virtuelle Maschine gestartet, führt sie die main-Methode der übergebenen Programmeinstiegsklasse aus. Die main-Methode hat stets denselben Aufbau: Sie muss den Namen `main` besitzen, darf keine Werte zurückliefern (angegeben durch das Schlüsselwort `void`) und muss genau einen formalen Parameter vom Typ `String[]` besitzen. Dass es sich um eine (statisch
static gebundene) Klassenmethode handelt, erkennt man am Schlüsselwort *static*.

```
public class Test {
    public static void main(String[] args) {
        Anweisungsblock
    }
}
```

Der formale Parameter der main-Methode (im obigen Beispiel `args` für „arguments“) referenziert ein String-Array, welches dazu dient, Argumente aufzunehmen, die dem Java-Programm ggf. beim Aufruf mitgegeben werden, und ist in der gesamten Methode ansprechbar.

Mit den in dieser Code-Skizze verwendeten Elementen der Sprache Java werden wir uns später noch intensiv beschäftigen. Vorläufig genügt es, zu wissen, dass wir an der mit *Anweisungsblock* gekennzeichneten Stelle eigene Anweisungen einfügen können, die sequentiell ausgeführt werden, wenn wir die Klasse Test als Einstiegsklasse eines Java-Programms verwenden[12].

Damit Ihre ersten praktischen Erfahrungen mit Java nicht allzu langweilig werden, führen wir nun noch drei Programmkonstrukte ein, die Sie einfach verwenden können, ohne dass wir sie an dieser Stelle bereits genauer erläutern wollen:

1. Um den Wert eines Ausdrucks auf der Standardausgabe auszugeben, können Sie die folgende Anweisung verwenden:

   `System.out.println(`*Ausdruck*`);`

[10] Im Absatz „Objektorientierte Programme“ von Abschnitt 2.1.2 „Klassen beschreiben Objekte“ wird eine differenziertere Sicht auf ein Java-Programm beschrieben, die auch Bibliotheksklassen einbezieht.

[11] Für verschiedene Rechnerarchitekturen und Betriebssysteme existieren unterschiedliche virtuelle Maschinen. Der Bytecode selbst ist unabhängig von der jeweiligen Rechnerumgebung.

[12] Wir haben hier die main-Methode und die Klasse Test als „public“ (= von überall her sichtbar) deklariert und werden das auch im Weiteren so handhaben. Eine genauere Erläuterung folgt in Lektion 2.

Auf der Standardausgabe wird eine textuelle Repräsentation des Ausdrucks ausgegeben, gefolgt von einem Zeilenumbruch[13]. Die Anweisung

```
System.out.print(Ausdruck);
```

bewirkt das Gleiche, nur dass kein Zeilenumbruch erzeugt wird.

2. Um aus einer Zeichenkette (Typ `String`), die aus einer Ziffernfolge besteht, die entsprechende Ganzzahl (`int`) zu erhalten, können Sie sich an folgendem Beispiel orientieren:

```
String s = "1234";
int i = Integer.parseInt(s);
```

3. Strings lassen sich mit dem Operator + untereinander und mit anderen Werten paarweise verketten. Genauer: Ist bei der Verwendung des +-Operators einer der beiden Operanden ein String, ist das Ergebnis wieder ein String. Dazu wird nötigenfalls der andere Operand automatisch zuerst in seine String-Repräsentation verwandelt, dann werden die beiden Strings aneinandergehängt. Die folgenden Anweisungen

```
String s = "Die Antwort lautet ";
int i = 42;
System.out.println(s + i + ".");
```

geben also den Text „Die Antwort lautet 42.“ auf der Standardausgabe aus.

Ad-hoc-Aufgabe 3

1. Erstellen Sie mit Hilfe der Entwicklungsumgebung Eclipse ein Programm, welches den Text „Hello World!“ auf der Standardausgabe ausgibt, übersetzen Sie es und führen Sie es aus.
2. Ändern Sie das Programm so ab, dass es Sie mit Ihrem Vor- und Nachnamen begrüßt. Den Namen und den Vornamen übergeben Sie dem Programm beim Programmstart als Argumente. Wie Sie die Argumente übergeben, wird im Vorkurs zum Modul 63611 (zu finden in Moodle) im Abschnitt „Erste Schritte mit Java und Eclipse“ gezeigt.
3. Was geschieht, wenn Sie dieses Programm starten und dabei statt Ihres Vor- und Nachnamens nur *ein* Argument übergeben?

1.2.1.6 Anweisungen, Blöcke und deren Ausführung

Anweisungen dienen dazu, den Kontrollfluss von Programmen zu definieren. Während Ausdrücke ausge*wert*et werden – üblicherweise mit Rückgabe eines Ergebniswertes –, spricht man bei Anweisungen von *Ausführung* (engl. *execution*).

[13] Es sei vorsorglich darauf hingewiesen, dass die textuelle Repräsentation vermutlich nicht in allen Fällen Ihren Vorstellungen entsprechen wird.

Dieser Unterabschnitt stellt die wichtigsten Anweisungen von Java vor und zeigt, wie aus Deklarationen und Anweisungen Blöcke gebildet werden können. Die zentralen Anweisungen „Zuweisung“ und „Methodenaufruf“ sind in Java sog. Ausdrucksanweisungen: Es handelt sich um Ausdrücke, die mit einem Semikolon abgeschlossen werden und dadurch als Anweisung fungieren. Dazu drei Beispiele:

Ausdrücke

```
i = 3;
flag = ( i >= 2 );
a.toString();
```

Der int-Variablen `i` wird 3 zugewiesen. Der booleschen Variable `flag` wird das Ergebnis des Vergleichs „`i` größer gleich `2`“ zugewiesen. Für das von der Variablen `a` referenzierte Objekt wird die Methode `toString` aufgerufen (gleichbedeutend dazu: Dem von der Variablen `a` referenzierten Objekt wird die Nachricht `toString` geschickt). Genauer werden wir auf den Methodenaufruf in Kap. 2 eingehen.

Blöcke

Eine in geschweifte Klammern eingeschlossene Sequenz von Variablendeklarationen und Anweisungen heißt in Java ein *Block* oder *Anweisungsblock*. Ein Block ist eine zusammengesetzte Anweisung. Das folgende Beispiel demonstriert insbesondere, dass Variablendeklarationen und Anweisungen gemischt werden dürfen:

```
{
   int i;
   i = 3;
   boolean flag;
   flag = (i >= 2);
   Object a = new Object();
   a.toString();
}
```

Variablendeklarationen lassen sich mit einer nachfolgenden Zuweisung zusammenfassen. Beispielsweise könnte man im obigen Beispiel die Deklaration von `flag` mit der Zuweisung in der nächsten Zeile wie folgt zu einer Anweisung verschmelzen: `boolean flag = (i >= 2);`
Variablendeklarationen in Blöcken sind ab der Deklarationsstelle bis zum Ende des Blockes gültig.

Ad-hoc-Aufgabe 4

Erstellen Sie mit Hilfe der Entwicklungsumgebung Eclipse ein Programm, in dessen main-Methode Sie einige Variablen verschiedener Typen deklarieren und diesen Variablen etwas zuweisen. Nehmen Sie dabei auch einige Zuweisungen vor, von denen Sie erwarten, dass der Compiler sie aufgrund der Typprüfung beim Kompilieren zurückweisen sollte. Schauen Sie sich an, wie Eclipse solche Typfehler markiert. Lesen Sie die von Eclipse generierten Fehlerbeschreibungen und versuchen Sie, deren Aussagen nachzuvollziehen.

1.2.1.7 Klassische Kontrollstrukturen.

Bedingte Anweisungen gibt es in Java in den Formen:

`if (`*boolescher_Ausdruck*`)` *Anweisung*

und

`if (`*boolescher_Ausdruck*`)` *Anweisung1* `else` *Anweisung2*

bedingte Anweisung

Bei der ersten Form wird *Anweisung* nur ausgeführt, wenn der boolesche Ausdruck zu `true` auswertet. Im zweiten Fall wird *Anweisung1* ausgeführt, wenn der boolesche Ausdruck zu `true` auswertet, andernfalls wird *Anweisung2* ausgeführt. Da die Anweisungen selbstverständlich wieder zusammengesetzte Anweisungen sein können, hat es sich weitgehend eingebürgert, sie immer als Anweisungsblöcke zu schreiben, auch wenn es sich um einfache Anweisungen handelt, also:

```
if (boolescher_Ausdruck) {
    Anweisung
}
```

bzw.

```
if (boolescher_Ausdruck) {
    Anweisung1
} else {
    Anweisung2
}
```

Damit hätten wir die Anweisung

```
value = newValue > value ? newValue : value;
```

aus Ad-hoc-Aufgabe 2, welche der Variablen `value` den größeren der beiden Werte der Variablen `value` und `newValue` zuweist, auch folgendermaßen schreiben können:

```
if (newValue > value) {
    value = newValue;
}
```

Für das Programmieren von Schleifen bietet Java die while- und die do-while-Anweisung sowie zwei Formen der for-Anweisung mit folgender Syntax:

```
while (boolescher_Ausdruck) Anweisung
do Anweisung while (boolescher_Ausdruck);
for (Init-Ausdruck ; boolescher_Ausdruck ; Ausdruck) Anweisung
for (Variablendeklaration : Ausdruck) Anweisung
```

Schleifen-Anweisung

Bei der while-Anweisung wird zuerst der boolesche Ausdruck ausgewertet. Liefert er den Wert *true*, wird die Anweisung im Rumpf der Schleife ausgeführt und die Ausführung der while-Anweisung beginnt von neuem. Andernfalls wird die Ausführung nach der while-Anweisung fortgesetzt.

Die do-while-Anweisung unterscheidet sich nur dadurch, dass der Schleifenrumpf auf jeden Fall einmal vor der ersten Auswertung des booleschen

Ausdrucks ausgeführt wird. Die do-while-Anweisung wird beendet, sobald der boolesche Ausdruck zu `false` auswertet.

Die Ausführung der for-Schleife in der ersten Form kann mit Hilfe der while-Schleife erklärt werden. Dazu betrachten wir ein kleines Programm, welches die Fakultät einer als Argument übergebenen Zahl berechnet (bei Eingaben größer als 20 wird allerdings der Wertebereich des Typs `long` überschritten, sodass das Programm nur im Bereich 0 bis 20 korrekt arbeitet):

```
public class Fakultaet {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        long result = 1;
        int i = 2;
        while (i <= n) {
            result = result * i;
            i = i + 1;
        }
        System.out.println(result);
    }
}
```

Äquivalent dazu ist folgendes Programm mit einer for-Schleife:

```
public class Fakultaet {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        long result = 1;
        for (int i = 2; i <= n; i = i + 1) {
            result = result * i;
        }
        System.out.println(result);
    }
}
```

Im Gegensatz zur while-Schleife können Deklaration und Initialisierung der Laufvariablen `i` bei der for-Schleife direkt im Schleifenkopf erfolgen. Diese Variable ist dann nur innerhalb der Schleife gültig.

Einer der klassischen Anwendungsfälle für die for-Schleife ist das Iterieren über ein Array. Das folgende Programmfragment legt ein Array mit zehn Elementen an und initialisiert diese in einer for-Schleife mit den Zahlen von 1 bis 10.

```
int[] arr = new int[10];
for (int i = 0; i < arr.length; i = i + 1) {
    arr[i] = i + 1;
}
```

for-each

Die zweite Form der `for`-Anweisung, auch *for-each* genannt, erlaubt das einfache Iterieren über Arrays oder Objekte, die vom Typ `java.lang.Iterable`[14] sind. Wir betrachten hier zunächst nur das Iterieren über Arrays. Statt über den Index der Elemente zu iterieren wie in der ersten Form, wird in der zweiten Form über die Elemente selbst iteriert.

[14] Ein Typ aus der Java-Standardklassenbibliothek, zu dem u. a. diverse Listen gehören

Es empfiehlt sich daher dringend, für die Variable, welche bei der Iteration einen Elementwert nach dem anderen aufnimmt, keinen Bezeichner zu wählen, der das Missverständnis fördern könnte, es handele sich bei dieser Variable um einen Index.

```
int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
// Lies: Für jedes Element im Array ...
for (int elem : arr) {
    System.out.print(elem + " ");
}
System.out.println();
```

Ad-hoc-Aufgabe 5

Die in Ad-hoc-Aufgabe 2 verwendete „unvollständige Initialisierung mehrdimensionaler Arrays“ wird insbesondere dann benötigt, wenn man mehrdimensionale Arrays erstellen will, die „nicht rechteckig sind“, sog. „Jagged Arrays“. Schreiben Sie ein Programm, welches mit Hilfe zweier geschachtelter for-Schleifen ein „zweidimensionales Zehner-int-Treppchen“ erzeugt und initialisiert, also ein int[ ][ ], dessen Elemente Referenzen auf int[ ]-Arrays mit zunehmender Länge (von 1 bis 10) sind.

Geben Sie *anschließend* (also nicht schon beim Erzeugen des Arrays!) mit zwei geschachtelten for-each-Schleifen die Elemente des „Treppchens“ so auf die Standardausgabe aus, dass dessen Struktur erkennbar wird. Eine hübsche Ausgabe könnte z. B. so aussehen:

```
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5
0 1 2 3 4 5 6
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8 9
```

Dabei entspricht jede Zeile einem Iterationsschritt über das „Array der ersten Dimension“, jede Zahl in einer Zeile einem Element eines „Arrays der zweiten Dimension“.

Diese Aufgabe ist nicht ganz einfach, deshalb noch einmal der Hinweis: Fragen Sie bei Problemen bitte im Diskussionsforum (Moodle) nach! Dort bekommen Sie ggf. erst einmal Erklärungen und kleinere Tipps, die Ihnen weiterhelfen, was wesentlich sinnvoller ist, als wenn Sie einfach nur in die Lösung schauen würden.

switch-Anweisung

Für die effiziente Implementierung von Fallunterscheidungen bietet Java die switch-Anweisung. Diese soll hier nur exemplarisch erläutert werden. Nehmen wir an, dass wir für den Wert einer int-Variablen `month` ausgeben wollen, wie viele Tage der entsprechende Monat hat. Ein typischer Fall für die switch-Anweisung:

```
switch (month) {
case 1:
case 3:
case 5:
case 7:
case 8:
case 10:
case 12:
    System.out.println("31 Tage");
    break;
case 2:
    System.out.println("28 oder 29 Tage");
    break;
case 4:
case 6:
case 9:
case 11:
    System.out.println("30 Tage");
    break;
default:
    System.out.print("Ungültiger Monat!");
    break;
}
```

Bei der Ausführung der switch-Anweisung wird zunächst der ganzzahlige Ausdruck[15] hinter dem Schlüsselwort `switch` ausgewertet, in obigem Fall wird also der Wert von `month` herangezogen. Sofern kein Fall für diesen Wert angegeben ist, wird die Ausführung nach dem Schlüsselwort `default` fortgesetzt. Andernfalls wird die Ausführung bei dem entsprechenden Fall fortgesetzt. Die break-Anweisungen im obigen Beispiel beenden die Ausführung des jeweiligen Falls und brechen die switch-Anweisung ab. Die Ausführung wird dann mit der Anweisung fortgesetzt, die der switch-Anweisung folgt. Existiert in einem Fall keine break-Anweisung, wird nach der Abarbeitung dieses Falls mit dem nächsten weitergemacht, bis die Ausführung entweder doch noch auf eine break-Anweisung trifft oder bis das Ende der switch-Anweisung erreicht ist.

break-Anweisung

Allgemein dient die break-Anweisung dazu, die umfassende Anweisung direkt zu verlassen; insbesondere kann sie auch zum Herausspringen aus Schleifen benutzt werden.

return-Anweisung

Entsprechend kann man mittels der return-Anweisung die Ausführung einer Methode oder eines Konstruktors (siehe Kapitel 2) sofort beenden. Syntaktisch tritt die return-Anweisung in zwei Varianten auf, je nachdem ob die zugehörige Methode ein Ergebnis an den Aufrufer der Methode zurückliefert oder nicht:

```
return Ausdruck       ;  // Wert des Ausdrucks liefert das Ergebnis
return;  // in als void deklarierten Methoden und in Konstruktoren
```

[15] Ab Java 5 ist es möglich, auch über Aufzählungstypen (Enums) zu switchen, ab Java 7 auch über Strings.

1.2.1.8 Abfangen von Ausnahmen

normale und abrupte Terminierung der Ausführung

Ebenso wie die Auswertung von Ausdrücken kann auch die Ausführung einer Anweisung normal oder abrupt terminieren. Bisher haben wir uns nur mit normaler Ausführung beschäftigt. Wie geht die Programmausführung aber weiter, wenn die Auswertung eines Ausdrucks oder die Ausführung einer Anweisung abrupt terminiert? Wird dann die Programmausführung vollständig abgebrochen? Die Antwort auf diese Fragen hängt von der betrachteten Programmiersprache ab. In Java gibt es spezielle Sprachkonstrukte, um abrupte Terminierung und damit *Ausnahmesituationen* zu behandeln: Mit der try-Anweisung können Programmierende aufgetretene Ausnahmen kontrollieren, mit der throw-Anweisung selbst eine abrupte Terminierung herbeiführen und damit eine Ausnahmebehandlung anstoßen.

try-Anweisung

Eine *try-Anweisung* dient dazu, Ausnahmen, die in einem Block auftreten, abzufangen und je nach dem Typ der Ausnahme zu behandeln. Die try-Anweisung hat folgende syntaktische Form:

```
try
  try-Block
catch (AusnahmeTyp Bezeichner) catch-Block_1
  ...
catch (AusnahmeTyp Bezeichner ) catch-Block_N
finally finally-Block
```

Die finally-Klausel ist optional, die try-Anweisung muss allerdings immer entweder mindestens eine catch-Klausel oder die finally-Klausel enthalten. Bevor wir die Bedeutung der try-Anweisung genauer erläutern, betrachten wir ein kleines Beispiel.

In der in Java vordefinierten Klasse `Integer` gibt es eine Methode `parseInt`, die eine Zeichenkette als Parameter entgegennimmt (genauer: Die Methode nimmt als Parameter eine Referenz auf ein String-Objekt entgegen). Stellt die Zeichenkette eine int-Konstante dar, terminiert die Methode normal und liefert den entsprechenden int-Wert als Ergebnis. Andernfalls terminiert sie abrupt und liefert ein Ausnahmeobjekt vom Typ `NumberFormatException`. Abbildung 1.5 zeigt an einem einfachen Beispiel, wie eine solche Ausnahme behandelt werden kann.

```
String str = "007L";
int m;
try {
    m = Integer.parseInt(str);
} catch(NumberFormatException e) {
    System.out.println("str ist kein int-Wert");
    m = 0;
}
System.out.println(m);
```

Abbildung 1.5: Programmfragment zur Behandlung einer Ausnahme

Da `parseInt` das Postfix „L“ beim eingegebenen Parameter nicht akzeptiert, wird die Ausführung des Methodenaufrufs innerhalb des try-Blocks abrupt

terminieren und eine Ausnahme vom Typ `NumberFormatException` erzeugen. Dies führt zur Ausführung der angegebenen catch-Klausel. Danach terminiert die gesamte try-Anweisung normal, sodass die Ausführung mit dem Aufruf von `println` in der letzten Zeile fortgesetzt wird.

Programme werden schnell unübersichtlich, wenn für jede elementare Anweisung, die möglicherweise eine Ausnahme erzeugt, eine eigene try-Anweisung programmiert wird. Stilistisch bessere Programme erhält man, wenn man die Ausnahmebehandlung am Ende größerer Programmteile zusammenfasst. Dies soll mit folgendem Programm illustriert werden, das seine beiden Argumente in int-Werte umwandelt und deren Quotienten berechnet:

```
public class Quotient {
    public static void main(String[] args) {
        try {
            int m = Integer.parseInt(args[0]);
            int n = Integer.parseInt(args[1]);
            int ergebnis = m / n;
            System.out.println(ergebnis);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Falsche Argument-Anzahl.");
        } catch (NumberFormatException e) {
            System.out.println("Mind. ein Argument ist kein int-Wert.");
        } catch (ArithmeticException e) {
            System.out.println("Division durch null.");
        } finally {
            System.out.println("Programmende.");
        }
    }
}
```

Eine Ausnahme vom Typ `ArrayIndexOutOfBoundsException` tritt bei Array-Zugriffen mit zu großem oder zu kleinem Index auf. Genau wie die NumberFormat-Ausnahme kann solch ein falscher Array-Zugriff innerhalb des obigen try-Blocks an zwei Stellen auftreten. In beiden Fällen wird die für diese Art von Exception vorgesehene catch-Klausel ausgeführt. Die unterste catch-Klausel fängt die mögliche Division durch null in der letzten Zeile des try-Blocks ab.

Die Ausführungssemantik für try-Anweisungen kann relativ komplex sein, da auch in den catch-Blöcken und dem finally-Block Ausnahmen auftreten können. Der Einfachheit halber gehen wir hier aber davon aus, dass die Ausführung der catch-Blöcke und des finally-Blocks normal terminiert. Unter dieser Annahme lässt sich die Ausführung einer try-Anweisung wie folgt zusammenfassen:

Führe zunächst den try-Block aus.

- Terminiert dessen Ausführung normal, führe den finally-Block aus. Die Ausführung der gesamten try-Anweisung terminiert in diesem Fall normal.

- Terminiert seine Ausführung abrupt mit einer Ausnahme *Exc*, suche nach der ersten zu *Exc* passenden catch-Klausel.

- Wenn es eine passende catch-Klausel gibt, führe den zugehörigen Block aus und danach den finally-Block. Die Ausführung der gesamten try-Anweisung terminiert in diesem Fall normal, d. h. die im try-Block aufgetretene Ausnahme wurde abgefangen und die Ausführung wird hinter der try-Anweisung fortgesetzt.
- Wenn es keine passende catch-Klausel gibt, führe den finally-Block aus. Die Ausführung der gesamten try-Anweisung terminiert in diesem Fall abrupt mit der Ausnahme *Exc*. Die Fortsetzung der Ausführung hängt dann vom Kontext der try-Anweisung ab: Ist sie in einer anderen try-Anweisung enthalten, übernimmt diese die Behandlung der Ausnahme *Exc*. Gibt es keine umfassende try-Anweisung, terminiert die umfassende Methode abrupt mit der Ausnahme *Exc*.

Dabei *passt* eine `catch`-Klausel zu einer geworfenen Ausnahme, wenn die Klasse, zu der das erzeugte Ausnahmeobjekt gehört, identisch mit oder eine Subklasse der Klasse ist, die zur Deklaration der zu fangenden Ausnahmen innerhalb der `catch`-Klausel verwendet wurde.

Die in den Beispielen vorkommenden Klassen `ArithmeticException`, `NumberFormatException` und `ArrayIndexOutOfBoundsException` sind alle Subklassen von `RuntimeException`, die selbst wiederum Subklasse der Klasse `Exception` ist. Die in Java vordefinierte Klassenhierarchie für Ausnahmetypen wird in Abschnitt 10.2.1 genauer eingeführt.

Selbstverständlich ist es auch möglich, eigene Ausnahmeklassen zu definieren. Alle selbst definierten Ausnahmeklassen müssen in Java direkt oder indirekt Subklassen der Klasse `Exception` sein. Die Deklaration eigener Ausnahmeklassen wird in Abschnitt 10.2 behandelt.

throw-Anweisung

Als letzte Anweisung behandeln wir die throw-Anweisung. Sie hat syntaktisch die Form:

`throw` *Ausdruck* `;`

Ausdruck muss hier ein Ausdruck sein, dessen Auswertung eine Referenz auf ein Ausnahmeobjekt (ein Objekt einer der Ausnahmeklassen) liefert. Die throw-Anweisung terminiert immer abrupt. Man verwendet sie, um selbst eine Ausnahmebehandlung auszulösen, zum Beispiel wenn man zu Beginn einer Anweisungsfolge feststellt, dass zu deren Ausführung nötige Bedingungen nicht eingehalten sind. So hätten wir etwa in unserem Fakultäts-Programm von Seite 32 wie folgt vorgehen können:

```
int n = Integer.parseInt(args[0]);
if (n > 20) throw new IllegalArgumentException();
long result = 1;
int i = 2;
while (i <= n) {
    result = result * i;
    i = i + 1;
}
System.out.println(result);
```

1.2.1.9 Ausnahmebehandlung und Nachrichtenversand

Die Möglichkeit, Ausnahmen auszulösen und abzufangen, spielt eine wichtige Rolle beim Nachrichtenversand, der ja das zentrale Konzept der objektorientierten Programmierung darstellt. Diesen Zusammenhang wollen wir uns anhand eines an die Realwelt angelehnten Beispiels schon jetzt etwas genauer ansehen:

Das Senden einer Nachricht an ein Objekt führt typischerweise dazu, dass dieses Objekt selbst wiederum Nachrichten versendet. Bei der Verarbeitung einer Nachricht entfernt man sich also leicht recht weit vom Ort des ursprünglichen Aufrufs.

Dabei werden oft Nachrichten an Objekte gesendet, die auf die Durchführung einer ganz bestimmten Teilaufgabe spezialisiert sind, diese aber in sehr unterschiedlichen Kontexten erledigen können.

Betrachten wir ein an die Realwelt angelehntes Beispiel: Eine Kundin bestellt bei einem Fahrradgeschäft ein Fahrrad mit einer besonderen Rahmenlackierung. Damit das Fahrradgeschäft das Rad an sie ausliefern kann, muss also ein Fahrradrahmen von einem Lackierer lackiert werden. Der Lackierer kauft den dazu benötigten Lack bei einem Großhändler, der ihn wiederum von einer Lackfabrik bezieht. Nun stellt die Lackfabrik aber nicht nur Fahrradlacke her, sondern Lacke für allerlei Anwendungszwecke. Manche davon lassen sich auch zum Lackieren höchst unterschiedlicher Endprodukte verwenden. Die Lackfabrik hat eine Methode „liefereLack“, über die man bei ihr unter Angabe einer Bestellnummer eine bestimmte Lacksorte anfordern kann. Diese Methode verwendet der Großhändler, um den Lack zu beziehen, den der Lackierer bei ihm zwecks Lackierung des Fahrrads bestellt hat.

Stellen wir uns nun vor, dass es in der Lackfabrik aus irgendwelchen Gründen Verzögerungen bei der Herstellung des vom Großhändler bestellten Lacks gab und dieser Lack deswegen zur Zeit nicht lieferbar ist. Die vom Großhändler aufgerufene Methode „liefereLack“ kann sich damit nicht regulär (unter Lieferung des Lacks) beenden, sondern es kommt bei ihrer Ausführung zu einem Fehler. Die Frage ist nun, wie dieser behandelt werden soll und durch wen. Eine denkbare sinnvolle Behandlung könnte etwa sein, die Kundin zu fragen, ob sie warten möchte, bis der gewünschte Lack wieder verfügbar ist, oder ob sie ihr Rad lieber schnell, dafür aber mit einer anderen Lackierung bekommen möchte.

Diese Behandlung kann aber nicht durch die Stelle vorgenommen werden, an der das Problem aufgetreten ist: Die Lackfabrik kennt den Kontext überhaupt nicht, in der ihr Lack verwendet werden soll, sie weiß schließlich weder etwas von dem bestellten Fahrrad, noch von der Kundin. Dieser Verlust des Ursprungskontexts ist eine zwangsläufige Folge der Tatsache, dass die Lackfabrik auf eine bestimmte Tätigkeit, nämlich das Herstellen von Lacken, spezialisiert ist.

Der Kontextverlust durch Spezialisierung trifft aber, wenn auch in unterschiedlichem Maße, auf *alle* Beteiligten zu, die an der Verarbeitung der ursprünglichen Nachricht beteiligt sind, mit der die Kundin ihr Fahrrad bestellt hat. Damit das Problem in der oben beschriebenen Weise behandelt werden kann, muss die entsprechende Information in der Nachrichtenverarbeitungs-

kette so lange „rückwärts“ durchgereicht werden, bis man an einer Stelle angekommen ist, an der genügend Information über den Kontext vorliegt, um zu entscheiden, wie weiter verfahren werden soll. Manchmal werden dazu wenige Schritte reichen, manchmal wird man bis zum Versender der allerersten Nachricht zurückgehen müssen.

Genau diese Aufgabe lässt sich mit Hilfe von Ausnahmen elegant bewältigen. Dazu führt der Lackhersteller in der Methode „liefereLack“ eine throw-Anweisung aus und erzeugt dabei ein passendes Ausnahmeobjekt. Da er die von ihm selbst ausgelöste Ausnahme nicht sinnvoll behandeln will und kann, wird sich die throw-Anweisung nicht in einer try-Anweisung mit zum Ausnahmeobjekt passender catch-Klausel befinden, weswegen sich die Methode „liefereLack“ abrupt beendet. Der Aufruf dieser Methode erfolgte aber aus einer Methode des Lackhändlers. Dieser hat nun also in *seiner* Methode, aus der heraus er der Lackfabrik eine Nachricht gesendet hat, eine Anweisung, die sich abrupt beendet hat (eben diesen Nachrichtenversand, also den Aufruf von „liefereLack“). Und da auch er sich nicht zuständig fühlt, befindet sich auch diese Anweisung nicht in einem try-Block mit passender catch-Klausel, weswegen sich die Methode des Lackhändlers ebenfalls abrupt beendet.

Auf diese Weise wird die Ausnahme immer weiter in der Kette der noch „offenen“ Nachrichten (die, deren Abarbeitung noch nicht beendet ist, weil sie ihrerseits erst einmal selbst wieder Nachrichten versendet haben und auf deren Abarbeitung „warten“) zurückgereicht, bis sie an einer Stelle ankommt, an der genug Kontext-Information vorhanden ist, um zu wissen, wie auf die Ausnahme reagiert werden soll. In einem Programm, welches unser Szenario sinnvoll umsetzt, wird sich an genau dieser Stelle ein try-Block mit einer zur Ausnahme passender catch-Klausel befinden.

Ad-hoc-Aufgabe 6

Sehen Sie sich die folgende Skizze einer Klasse mit Ausnahmebehandlung an. Ist die skizzierte Vorgehensweise korrekt? Wenn nein, was würden Sie ändern?

```
public class ExceptionTest {
    public static void main(String[] args) {
        try {
            // allerlei Anweisungen die verschiedene
            // Exceptions ausloesen können
        } catch (Exception e) {
            // Behandlung
        } catch (ArrayIndexOutOfBoundsException e) {
            // Behandlung
        } catch (NumberFormatException e) {
            // Behandlung
        }
    }
}
```

1.2.2 Objektorientierte Programmierung mit Java

Dieser Abschnitt führt in die programmiersprachliche Realisierung objektorientierter Konzepte ein. Anhand eines kleinen Beispiels demonstriert er, wie die zentralen Aspekte des objektorientierten Grundmodells, das in Abschn. 1.1.1 vorgestellt wurde, mit Hilfe von Java umgesetzt werden können. Insgesamt verfolgt dieser Abschnitt die folgenden Ziele:

1. Das objektorientierte Grundmodell soll zusammengefasst und konkretisiert werden.
2. Die Benutzung objektorientierter Konzepte im Rahmen der Programmierung mit Java soll demonstriert werden.
3. Die Einführung in die Sprache Java soll fortgesetzt werden.

Als Beispiel verwenden wir einen kleinen Ausschnitt der Modellierung von Personengruppen an einer Universität, skizziert in Abb. 1.6. Wir definieren Personen- und Studierenden-Objekte und statten sie zur Illustration mit einfachen Methoden aus. Jedes Objekt der Klasse Person hat eine Methode, um den Namen und das Geburtsdatum zu erfragen. Studierende können darüber hinaus nach Matrikelnummer und Semesterzahl befragt werden. Bei den Angestellten kommen stattdessen Angaben über das Arbeitsverhältnis und die Zuordnung zu Untergliederungen der Universität hinzu. Wir zeigen, wie das Erzeugen von Objekten realisiert und wie mit Objekten programmiert werden kann.

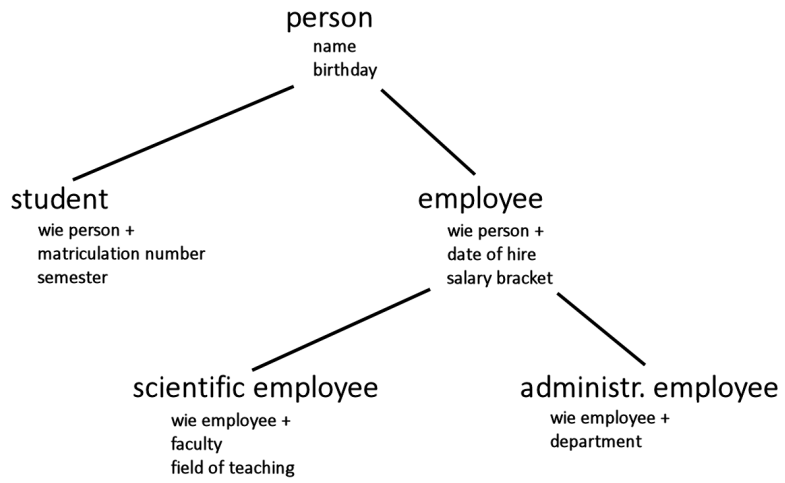

Abbildung 1.6: Klassifikation der Personen an einer Universität

1.2.2.1 Objekte, Klassen, Methoden, Konstruktoren

Objekte besitzen einen Zustand und Methoden, mit denen sie auf Nachrichten reagieren. Programmtechnisch wird der Zustand durch mehrere Attribute realisiert, also objektlokale Variablen. Solche objektlokalen Variablen werden ebenso wie die Methoden in Java im Rahmen von Klassen deklariert. Man beschreibt also nicht für ein einzelnes Objekt, welche Attribute und Methoden

es hat, sondern man beschreibt für eine Klasse, welche Attribute und Methoden alle zu dieser Klasse gehörenden Objekte haben[16]. Bei der Erzeugung eines Objekts werden diese Variablen dann initialisiert. Person-Objekte mit den in Abb. 1.6 angegebenen Attributen `name` und `birthday` werden spezifiziert durch eine Klasse `Person` (Die Datumsangabe codieren wir hier zur Vereinfachung als int-Zahl im Format JJJJMMTT):

```
public class Person {
    String name;
    int birthday; /* in der Form JJJJMMTT */
}
```

Objekte der Klasse `Person` kann man durch Aufruf des sog. *Default-Konstruktors* `new Person()` erzeugen. Durch das Schlüsselwort `new` wird ein Speicherbereich für das neue Objekt alloziert, durch den Konstruktor werden die Attribute des Objekts mit Standardwerten initialisiert. Möchte man die Initialisierungswerte selbst festlegen (was der Normalfall ist), muss man der Klassendeklaration einen selbst geschriebenen *Konstruktor* hinzufügen:

Konstruktor

```
public class Person {
    String name;
    int birthday; /* in der Form JJJJMMTT */

    Person(String name, int birthday) {
        this.name = name;
        this.birthday = birthday;
    }
}
```

Der Name eines Konstruktors muss in Java mit dem Namen der Klasse übereinstimmen, zu der er gehört. Ein Konstruktor kann Parameter besitzen, die zur Initialisierung der Attribute verwendet werden können (s.o.). Da wir für die zu initialisierenden Attribute und die Parameter, deren Wert zur Initialisierung verwendet wird, den gleichen Bezeichner verwendet haben[17], müssen wir deutlich machen, dass auf der linken Seite der Zuweisung das Attribut gemeint ist. Dies geschieht in der oben gezeigten Weise. Das Schlüsselwort *this* steht dabei für „das Objekt, in dem wir uns gerade befinden“, `this.name` bezeichnet also das Attribut `name` dieses Objekts. Näheres zu Konstruktoren finden Sie in Abschnitt 2.1.2 und in Abschnitt 6.2.

Wir statten Person-Objekte nun mit zwei Methoden aus: Erhält eine Person die Nachricht `print`, soll sie ihren Namen und ihr Geburtsdatum ausgeben. Erhält eine Person die Nachricht `isBirthday` mit einem Datum als Parameter, soll sie prüfen, ob sie an diesem Datum Geburtstag hat. Die bisherige Klassendeklaration von `Person` wird also ergänzt um die Methoden `print` und `isBirthday`:

[16] Objektorientierte Sprachen, bei denen dies so ist, bezeichnet man als *klassenbasierte* Sprachen. Die meisten gängigen OO-Sprachen sind klassenbasiert.

[17] Das ist eine in Java übliche Vorgehensweise: Wenn man für ein Attribut einen gut passenden Bezeichner gefunden hat, erscheint es unnötig, für den Parameter, der eben dieses Attribut initialisiert, also für genau denselben Sachverhalt steht, einen zusätzlichen abweichenden Bezeichner zu verwenden.

```
public class Person {
    String name;
    int birthday; /* in der Form JJJJMMTT */

    Person(String name, int birthday) {
        this.name = name;
        this.birthday = birthday;
    }

    void print() {
        System.out.println("Name: " + this.name);
        System.out.println("Geburtsdatum: " + this.birthday);
    }

    boolean isBirthday(int date) {
        return birthday % 10000 == date % 10000;
        // Das Prozentzeichen ist der Modulo-Operator, welcher
        // den Rest einer ganzzahligen Division liefert.
    }
}
```

Die Methode `print` gibt die beiden Attribute `name` und `birthday` des Person-Objekts aus, auf dem sie aufgerufen wird. Dieses Person-Objekt wird der Methode bei ihrem Aufruf als impliziter Parameter mitgegeben. Dieser Parameter wird häufig als *this-Objekt* oder *self-Objekt* bezeichnet. In der Methode `print` werden die Attribute wie schon bei der Inititaliserung im Konstruktor explizit mit Benennung dieses Objekts angesprochen. Die Methode `isBirthday` demonstriert, dass man die explizite Nennung des this-Objekts auch weglassen kann, wenn eindeutig ist, dass dessen Attribute gemeint sind.

this-Objekt

Werden einer Methode neben dem impliziten *this*-Parameter noch weitere Parameter übergeben, so werden diese zur besseren Unterscheidung auch *formale Parameter* genannt. In `isBirthday` wird also der Wert des formalen Parameters `date` mit der Monats- und Tagesangabe des this-Objekts verglichen.

formaler Parameter

1.2.2.2 Spezialisierung und Vererbung

Gemäß Abb. 1.6, S. 40, sind Studierende Personen mit zwei zusätzlichen Attributen. Studierende sind also spezielle Personen. Es wäre demnach wünschenswert, die Implementierung von Student-Objekten durch geeignete Erweiterung und Modifikation der Implementierung von Person-Objekten zu erhalten.

Objektorientierte Programmiersprachen wie Java unterstützen es, Klassen wie gewünscht zu erweitern. Dass eine Klasse eine Erweiterung einer anderen Klasse (deren Subklasse) ist, wird mit dem Schlüsselwort `extends` ausgedrückt.

Da eine Klasse in Java eine spezielle Art Typ ist, wird durch dieses *Subclassing* gleichzeitig festgelegt, dass der Typ der Subklasse Subtyp des Typs der Superklasse ist. Die erweiternde Klasse erbt die Attribute und Methoden ihrer Superklasse, im Beispiel die Attribute `name` und `birthday` und die Methode `isBirthday`.

Subclassing

```
public class Student extends Person {
    int matriculationNr;
    int semester;

    Student(String name, int birthday, int matriculationNr, int semester) {
        super(name, birthday);
        this.matriculationNr = matriculationNr;
        this.semester = semester;
    }

    void print() {
        super.print();
        System.out.println("Matrikelnr: " + matriculationNr);
        System.out.println("Semesterzahl: " + semester);
    }

    int getMatriculationNr() {
        return matriculationNr;
    }
}
```

Anhand dieses Beispiels lassen sich einige Aspekte erkennen, die für die Vererbung wichtig sind.

Die speziellere Klasse (hier `Student`) erbt die Attribute und Methoden der allgemeineren Klasse (hier `Person`). Ihre Instanzen verstehen also mindestens die gleichen Nachrichten (hier `print` und `isBirthday`) wie die Instanzen der Superklasse. Mit Hilfe von in der Subklasse neu hinzugefügten Methoden können sie ggf. zusätzlich Nachrichten verarbeiten, welche die Instanzen der Superklasse nicht verstehen (hier `getMatriculationNr`).

Da die Methode `isBirthday` auch für Student-Objekte korrekt funktioniert, kann sie unverändert übernommen werden. Etwas anders sieht es mit der Methode `print` aus: Wenn einem Student-Objekt die Nachricht `print` geschickt wird, sollen natürlich nicht nur die geerbten Attribute `name` und `birthday` ausgegeben werden, sondern auch `semester` und `matriculationNr`.

Java sieht deshalb die Möglichkeit vor, eine geerbte Methode in der Subklasse durch eine neue Version zu ersetzen. Man bezeichnet dies als *Überschreiben* der Methode der Superklasse. Oft wird man aber – wie in unserem Fall – die geerbte Methode gar nicht komplett ersetzen, sondern lediglich um zusätzliche Anweisungen ergänzen wollen. Mit Hilfe eines kleinen „Tricks“ ist auch das möglich. Dazu wird die von der Superklasse geerbte Methode zwar ersetzt, jedoch ruft man aus der neuen Methode mit Hilfe des Schlüsselworts `super` die ursprüngliche Version auf. In unserem Beispiel wird in der Methode `print` der Klasse `Student` durch den Aufruf von `super.print();` die Methode `print` der Klasse `Person` aufgerufen und dazu genutzt, die Attribute `name` und `birthday` auszugeben. Anschließend folgt dann die Ausgabe von `semester` und `matriculationNr`.

Überschreiben

Ähnlich verhält es sich mit dem Konstruktor: Zwar werden Konstruktoren in Java nicht vererbt (den Grund dafür werden wir später noch kennen lernen), können also auch nicht überschrieben werden. Es ist aber oft wünschenswert, in der Superklasse bereits codierte Initialisierungen auch aus

Konstruktoren der Subklasse auszuführen. Dies erfolgt hier, indem in der ersten Zeile des Subklassenkonstruktors mittels `super(name, birthday);` der Konstruktor der Klasse `Person` aufgerufen wird, der dann die Initialisierung der entsprechenden Instanzvariablen übernimmt.

1.2.2.3 Subtyping und dynamisches Binden

Da Student-Objekte alle Operationen unterstützen, die man auf Person-Objekte anwenden kann (alle Nachrichten verstehen, welche Person-Objekte verstehen), können Student-Objekte prinzipiell an allen Programmstellen verwendet werden, an denen Person-Objekte zulässig sind. In der objektorientierten Programmierung gestattet man es generell, Objekte von spezielleren Typen überall dort zu verwenden, wo Objekte von allgemeineren Typen zulässig sind. Ist S ein speziellerer Typ als T, so wird S auch als *Subtyp* von T bezeichnet. `Student` ist also ein Subtyp von `Person`.

Subtyp

Der entscheidende Vorteil von Subtyping ist, dass wir einen Algorithmus, der für einen Typ formuliert ist, für Objekte aller Subtypen verwenden können. Um dies an unserem Beispiel zu demonstrieren, wollen wir alle Elemente eines Arrays vom Typ `Person[]` drucken. Das folgende Programm zeigt, wie einfach das mit objektorientierten Techniken geht:

```
public class PersonPrintTest {
    public static void main(String[] args) {
        Person[] persons = new Person[3];
        persons[0] = new Person("Meyer", 19831007);
        persons[1] = new Student("Müller", 19641223, 6758475, 5);
        persons[2] = new Student("Planck", 18580423, 3454545, 47);

        for (int i = 0; i < persons.length; i = i + 1) {
            persons[i].print();
            System.out.println();
        }
    }
}
```

Im Rumpf der main-Methode wird zunächst ein Personen-Array erzeugt und mit Referenzen auf ein Person-Objekt und zwei Student-Objekte initialisiert. Die Zuweisungen sind gültig, denn jedes Student-Objekt *ist* auch ein Person-Objekt. In der for-Schleife muss nun nur die Druckmethode für jedes Element aufgerufen werden. Ist das Element ein Person-Objekt, wird automatisch die Methode `print` der Klasse `Person` ausgeführt; ist es ein Student-Objekt, wird die Methode `print` der Klasse `Student` ausgeführt. Eine explizite Unterscheidung zwischen Studierenden und Personen ist *nicht* nötig.

Zur Erinnerung: Dies entspricht dem Grundprinzip der objektorientierten Programmierung, nachdem jedes Objekt selbst dafür zuständig ist, wie es auf den Empfang einer Nachricht reagiert. Um dieses Prinzip umzusetzen, müssen Methodenaufrufe dynamisch gebunden werden[18], siehe Abschnitt 1.1.2.2.

[18] Es gibt Ausnahmen. Das sind z.B. die schon erwähnten Klassenmethoden, welche ja nicht zu einem Objekt gehören, aber auch Methoden, die als `private` oder `final` deklariert sind und daher in Subklassen nicht überschrieben werden können (wird in Abschnitt 6.6 behandelt).

Ad-hoc-Aufgabe 7

Vorbemerkung: In dieser Aufgabe werden Sie erstmals ein Programm erstellen, das aus *mehreren* von Ihnen geschriebenen Klassen besteht. Legen Sie bitte jede Klasse in einer eigenen Übersetzungseinheit an. Wenn Sie Eclipse verwenden, erstellen Sie dazu eine neue Klasse, indem Sie aus dem Menü „File“ den Menüpunkt „New – Class“ verwenden.

Zweite Vorbemerkung: Diese Aufgabe ist vergleichsweise schwierig. Um sie zu lösen, müssen Sie einiges, was Sie aus der imperativen Programmierung kennen, mit dem zusammenbringen, was Sie in dieser Lektion gelesen haben. Dafür ist sie aber gut geeignet, häufig vorkommende Lücken und Missverständnisse aufzudecken.

Falls Sie die Aufgabe nicht auf Anhieb alleine lösen können, ist das also völlig normal und kein Grund für Frustration. Schauen Sie dann bitte *nicht* direkt in die Musterlösung, sondern versuchen Sie, im Diskussionsforum zu beschreiben, wie weit sie gekommen sind, und wo es hakt. Dort bekommen Sie dann passende Hinweise, mit denen Sie selbst weiterkommen.

1. Schreiben Sie eine Klasse `ArrayAddressBook` für einfache Adressbücher mit folgenden Eigenschaften:
   - Ein Adressbuch besitzt ein Attribut `persons`, welches vom Typ `Person[]` ist (wir verwenden hier die Klasse `Person` von Seite 42). Dieses Attribut soll im Konstruktor des Adressbuchs mit einem neu erzeugten Array initialisiert werden, wobei die Größe des Arrays dem Konstruktor als Parameter übergeben wird. Wird eine Größe kleiner als 1 übergeben, soll 1 als Größe verwendet werden.
   - Das Adressbuch hat eine Methode `addPerson()`, welche eine Referenz auf ein Person-Objekt als Parameter akzeptiert und die übergebene Person an die erste noch freie Stelle in das Array `persons` schreibt. Das Array wird also „von unten“ (beginnend mit Index 0) befüllt. Um die Einfügestelle ermitteln zu können, bekommt das Adressbuch ein Attribut, welches den Index des nächsten noch unbelegten Array-Elements repräsentiert. Dieses Attribut muss natürlich bei Veränderungen aktualisiert werden.
   - Damit die Anzahl der Personen, die aufgenommen werden können, nicht begrenzt ist, soll beim Hinzufügen einer Person mit `addPerson()` zunächst überprüft werden, ob das Array bereits voll belegt ist. Wenn ja, soll ein neues Array doppelter Größe angelegt und das alte Array dort hineinkopiert werden. Das Verdoppeln und Umkopieren soll in eine eigene Methode ausgelagert werden.
   - Um das Adressbuch sinnvoll testen zu können, bekommt es eine Methode `print()`. Diese gibt zunächst eine Zeile aus, in der steht, wie viele Einträge zur Zeit im Adressbuch sind, dann eine, in der steht, wie groß seine Kapazität zur Zeit ist. Anschließend ruft sie auf allen im Adressbuch enthaltenen Person-Objekten deren Methode print() auf.
2. Schreiben Sie eine Klasse `AddressBookTest` mit einer main-Methode. In dieser erzeugen Sie ein `ArrayAddressBook` mit Anfangsgröße 3. Erzeugen Sie dann einige Person-Objekte und legen Sie sie mit der Methode `addPerson()` ins Adressbuch. Um zu überprüfen, ob das Einfügen und das automatische Verdoppeln der Kapazität funktioniert, rufen Sie an geeigneten Stellen die Methode `print()` des Adressbuchs auf.
3. Erstellen Sie weitere Subklassen von Person und testen Sie Ihr Adressbuch auch mit Instanzen dieser Klassen. Sie können sich dabei an Abb. 1.6 orientieren. Versuchen Sie, mehrfach vorkommenden gleichen Code zu vermeiden, indem Sie die gezeigten Möglichkeiten des „Ergänzens“ von Methoden und des Aufrufs von Superklassenkonstruktoren aus Subklassen einsetzen.

1.2.3 Objektorientierte Sprachen im Überblick

Objektorientierte Sprachen unterscheiden sich darin, wie sie das objektorientierte Grundmodell durch Sprachkonstrukte unterstützen und wie sie die objektorientierten Konzepte mit anderen Konzepten verbinden. Mittlerweile gibt es eine Vielzahl von Sprachen, die objektorientierte Aspekte unterstützen. Einige von ihnen sind durch Erweiterung existierender Sprachen entstanden, beispielsweise C++ und Objective C (als Erweiterung von C). Andere Sprachen wurden speziell für die objektorientierte Programmierung entwickelt, beispielsweise Simula67 , Smalltalk, Eiffel und Java. Außerdem gibt es etliche Programmiersprachen, die mehrere Programmierparadigmen unterstützen, also z. B. Elemente der objektorientierten, prozeduralen und funktionalen Programmierung enthalten.

An dieser Stelle des Lehrtextes fehlen noch die Voraussetzungen, um die Unterschiede zwischen objektorientierten Sprachen detailliert zu behandeln. Andererseits ist es wichtig, einen Überblick über die Variationsbreite bei der sprachlichen Umsetzung objektorientierter Konzepte zu besitzen, bevor man sich auf eine bestimmte Realisierung – in unserem Fall auf Java – einlässt. Denn nur die Kenntnis der unterschiedlichen Umsetzungsmöglichkeiten erlaubt es, zwischen den allgemeinen Konzepten und einer bestimmten Realisierung zu trennen. Deshalb werden schon hier Gemeinsamkeiten und Unterschiede bei der sprachlichen Umsetzung übersichtsartig zusammengestellt, auch wenn die verwendeten Begriffe erst im Laufe des Lehrtextes genauer erläutert werden.

Gemeinsamkeiten. In fast allen objektorientierten Sprachen wird ein Objekt als ein Verbund von Variablen und Methoden realisiert, d. h. so, wie wir es in Abschn. 1.1.2.2 erläutert haben. Allerdings werden die Methoden implementierungstechnisch in den meisten Fällen nicht den Objekten zugeordnet, sondern den Klassen.

Eine verbreitete Gemeinsamkeit objektorientierter Sprachen ist die *synchrone* Kommunikation, d. h. der Aufrufer einer Methode kann erst fortfahren, wenn die Methode terminiert hat. Mit dem Vokabular des objektorientierten Grundmodells formuliert, heißt das, dass Senderobjekte nach dem Verschicken einer Nachricht warten müssen, bis die zugehörige Methode vom Empfänger abgearbeitet wurde. Erst nach der Bearbeitung und ggf. nach Empfang eines Ergebnisses kann der Sender seine Ausführung fortsetzen. Das objektorientierte Grundmodell ließe auch andere Kommunikationsarten zu. Beispiele dafür findet man vor allem bei objektorientierten Sprachen und Systemen zur Realisierung verteilter Anwendungen (vgl. Kap. 14).

Unterschiede. Die Unterschiede zwischen objektorientierten Programmiersprachen sind zum Teil erheblich. Die folgende Liste fasst die wichtigsten Unterscheidungskriterien zusammen:

- Objektbeschreibung: In den meisten Sprachen werden Objekte durch sogenannte Klassendeklarationen beschrieben. Andere Sprachen verzichten auf Klassen und bieten Konstrukte an, mit denen man existie-

rende Objekte während der Programmlaufzeit klonen kann und dann Attribute und Methoden hinzufügen bzw. entfernen kann. Derartige Sprachen nennt man *prototypbasiert* (Beispiele sind die Sprachen Self und JavaScript) .

- Reflexion: Sprachen unterscheiden sich darin, ob Klassen und Methoden auch als Objekte realisiert sind, d. h. einen Zustand besitzen und Empfänger und Sender von Nachrichten sein können. Beispielsweise sind in Smalltalk Klassen und sogar Codeblöcke vollwertige Objekte, und in BETA sind Methoden als Objekte modelliert.

- Typsysteme: Die Typsysteme objektorientierter Sprachen sind sehr verschieden. Am einen Ende der Skala stehen untypisierte Sprachen (z. B. Smalltalk), am anderen Ende Sprachen mit strenger Typprüfung, Subtyping und Generizität (z. B. Java, C# und Eiffel).

- Vererbung: Auch bei der Vererbung von Programmteilen zwischen Klassen gibt es wichtige Unterschiede. In vielen Sprachen, z. B. in Java, Smalltalk oder C#, kann eine Klasse nur von *einer* anderen Klasse erben (Einfachvererbung), andere Sprachen ermöglichen Mehrfachvererbung (z. B. CLOS, C++ und Eiffel). Meist sind Vererbung und Subtyping eng aneinander gekoppelt. Es gibt aber auch Sprachen, die diese Konzepte sauber voneinander trennen (beispielsweise Sather). Variationen gibt es auch dabei, welche Programmteile vererbt werden können.

- Spezialisierung: Objektorientierte Sprachen bieten Sprachkonstrukte an, um geerbte Methoden zu spezialisieren. Diese Sprachkonstrukte unterscheiden sich zum Teil erheblich voneinander.

- Kapselung: Kapselungskonstrukte sollen Teile der Implementierung vor Benutzenden verbergen. Die meisten OO-Sprachen unterstützen Kapselung in der einen oder anderen Weise.

- Strukturierungskonstrukte: Ähnlich der uneinheitlichen Situation bei der Kapselung gibt es eine Vielfalt von Lösungen zur Strukturierung einer Menge von Klassen. Einige Sprachen sehen dafür Modulkonzepte vor (z. B. Modula-3), andere benutzen die Schachtelung von Klassen zur Strukturierung.

- Implementierungsaspekte: Ein wichtiger Implementierungsaspekt objektorientierter Programmiersprachen ist die Frage, wer den Speicher für die Objekte verwaltet. Beispielsweise sind in C++ die Programmierenden dafür zuständig, während Java und C# eine automatische Speicherverwaltung anbieten.

Die Liste erhebt keinen Anspruch auf Vollständigkeit. Beispielsweise unterscheiden sich objektorientierte Sprachen auch in den Mechanismen zur dynamischen Bindung, in der Unterstützung von Parallelität und Verteilung sowie in den Möglichkeiten zum dynamischen Laden und Verändern von Programmen zur Laufzeit.