David J. Barnes • Michael Kölling

Java lernen mit BlueJ

Objects first – Eine Einführung in Java

6., aktualisierte Auflage

Java lernen mit BlueJ

David J. Barnes • Michael Kölling

Java lernen mit BlueJ

Objects first - Eine Einführung in Java

6., aktualisierte Auflage

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Die Informationen in diesem Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt. Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen. Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Autor dankbar.

Authorized translation from the English language edition, entitled OBJECTS FIRST WITH JAVA – A PRACTICAL INTRODUCTION USING BLUEJ, 6th Edition by DAVID BARNES and MICHAEL KÖLLING, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2017 Pearson Education Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

GERMAN language edition published by PEARSON DEUTSCHLAND GMBH, Copyright © 2017.

Fast alle Produktbezeichnungen und weitere Stichworte und sonstige Angaben, die in diesem Buch verwendet werden, sind als eingetragene Marken geschützt. Da es nicht möglich ist, in allen Fällen zeitnah zu ermitteln, ob ein Markenschutz besteht, wird das ®-Symbol in diesem Buch nicht verwendet.

10 9 8 7 6 5 4 3 2 1

20 19 18 17

ISBN 978-3-86894-911-7 (Buch)
ISBN 978-3-86326-961-6 (E-Book)

© 2017 by Pearson Deutschland GmbH
Lilienthalstraße 2, 85399 Hallbergmoos/Germany
Alle Rechte vorbehalten
www.pearson.de
A part of Pearson plc worldwide

Programmleitung: Birger Peil, bpeil@pearson.de
Übersetzung und Korrektorat: Katharina Pieper, Berlin, pieper.katharina@googlemail.com
Coverillustration: Blue jay image. Reprinted with permission copyright Suman Roy Choudhury@sqr-photo@yahoo.com
Herstellung: Claudia Bäurle, cbaeurle@pearson.de
Satz: inpunkt[w]o, Haiger (www.inpunktwo.de)
Druck- und Verarbeitung: DZS-Grafik, d.o.o., Ljubljana
Printed in Slovenia

Inhaltsverzeichnis

Vorwort von James Gosling, Erfinder von Java 13
Vorwort für den Lehrenden 14
Danksagungen 25
Projekte, die in diesem Buch detailliert besprochen werden 26

Teil I Objekte und Klassen 29

Kapitel 1 Objekte und Klassen 31
1.1 Objekte und Klassen 31
1.2 Instanzen erzeugen 32
1.3 Methoden aufrufen 33
1.4 Parameter 35
1.5 Datentypen 36
1.6 Eine Klasse, viele Instanzen 37
1.7 Zustand 38
1.8 Das Innenleben eines Objekts 38
1.9 Java-Code 40
1.10 Objektinteraktion 41
1.11 Quelltext 42
1.12 Ein weiteres Beispiel 44
1.13 Aufrufergebnisse 44
1.14 Objekte als Parameter 45

Kapitel 2 Klassendefinitionen 51
2.1 Ticketautomaten 51
2.2 Eine Klassendefinition untersuchen 53
2.3 Der Kopf der Klasse 55
2.4 Datenfelder, Konstruktoren und Methoden 56
2.5 Datenübergabe mit Parametern 62
2.6 Zuweisungen 64
2.7 Methoden 65
2.8 Sondierende und verändernde Methoden 67

2.9 Ausgaben in Methoden 70
2.10 Zusammenfassung der Methoden 72
2.11 Zusammenfassung des naiven Ticketautomaten 72
2.12 Bewertung des Entwurfs des naiven Ticketautomaten 73
2.13 Entscheidungen treffen: die bedingte Anweisung 76
2.14 Ein weiteres Beispiel für eine bedingte Anweisung 78
2.15 Hervorhebung von Sichtbarkeitsbereichen 79
2.16 Lokale Variablen 80
2.17 Datenfelder, Parameter und lokale Variablen 82
2.18 Zusammenfassung des besseren Ticketautomaten 84
2.19 Übungen zur Selbstüberprüfung 84
2.20 Vertrautes neu betrachtet 86
2.21 Methoden aufrufen 88
2.22 Ausdrücke testen: die Direkteingabe 89

Kapitel 3 Objektinteraktion 97
3.1 Das Uhren-Beispiel 97
3.2 Abstraktion und Modularisierung 98
3.3 Abstraktion in Software 99
3.4 Modularisierung im Uhren-Beispiel 100
3.5 Implementierung der Uhrenanzeige 100
3.6 Klassendiagramme und Objektdiagramme 101
3.7 Primitive Typen und Objekttypen 103
3.8 Die Klasse Nummernanzeige 103
3.9 Die Klasse Uhrenanzeige 110
3.10 Objekte erzeugen Objekte 112
3.11 Mehrere Konstruktoren 114
3.12 Methodenaufrufe 114
3.13 Ein weiteres Beispiel für Objektinteraktion 118
3.14 Die Benutzung eines Debuggers 122
3.15 Mehr zu Methodenaufrufen 126

Kapitel 4 Objektsammlungen 131
4.1 Themen aus Kapitel 3 vertiefen 131
4.2 Die Abstraktion Sammlung 132
4.3 Ein Verwaltungssystem für Musikdateien 133
4.4 Eine Bibliotheksklasse verwenden 135
4.5 Objektstrukturen mit Sammlungen 138
4.6 Generische Klassen 139
4.7 Nummerierung in Sammlungen 140
4.8 Musikdateien abspielen 144

4.9 Komplette Sammlungen verarbeiten 146
4.10 Unbestimmte Iteration 152
4.11 Verbesserung der Struktur – die Klasse Track 159
4.12 Der Typ Iterator 163
4.13 Zusammenfassung des Musiksammlung-Beispiels 167
4.14 Ein weiteres Beispiel: ein Auktionssystem 169

Kapitel 5 Funktionale Verarbeitung von Sammlungen (fortgeschrittene Konzepte) 183
5.1 Ein neuer Blick auf die Themen von Kapitel 4 183
5.2 Monitoring von Tierpopulationen 185
5.3 Lambda-Ausdrücke – ein erster Blick 188
5.4 Die forEach-Methode von Sammlungen 190
5.5 Streams 193

Kapitel 6 Bibliotheksklassen nutzen 207
6.1 Die Dokumentation der Bibliotheksklassen 208
6.2 Das Kundendienstsystem 209
6.3 Die Klassendokumentation lesen 215
6.4 Zufälliges Verhalten einbringen 221
6.5 Pakete und Importe 226
6.6 Benutzung von Map-Klassen für Abbildungen 227
6.7 Der Umgang mit Mengen 232
6.8 Zeichenketten zerlegen 233
6.9 Abschluss des Kundendienstsystems 234
6.10 Autoboxing und Wrapper-Klassen 236
6.11 Die Klassendokumentation schreiben 239
6.12 Öffentliche und private Eigenschaften 242
6.13 Klassen über ihre Schnittstelle verstehen 245
6.14 Klassenvariablen und Konstanten 250
6.15 Klassenmethoden 253
6.16 Programmausführung ohne BlueJ 254
6.17 Weitere fortgeschrittene Konzepte 255

Kapitel 7 Sammlungen mit fester Größe – Arrays 263
7.1 Sammlungen fester Größe 263
7.2 Arrays 263
7.3 Die Analyse einer Logdatei 264
7.4 Die for-Schleife 270
7.5 Das Projekt Automat 276
7.6 Arrays mit mehr als einer Dimension (fortgeschritten) 285
7.7 Arrays und Streams (fortgeschritten) 291

Kapitel 8 Klassenentwurf 293
8.1 Einführung 294
8.2 Die Welt von Zuul 296
8.3 Kopplung und Kohäsion 297
8.4 Code-Duplizierung 298
8.5 Erweiterungen für Zuul 301
8.6 Kopplung 303
8.7 Entwurf nach Zuständigkeiten 308
8.8 Änderungen lokal halten 311
8.9 Implizite Kopplung 311
8.10 Vorausdenken 315
8.11 Kohäsion 316
8.12 Refactoring 320
8.13 Refactoring für Sprachunabhängigkeit 324
8.14 Entwurfsregeln 330

Kapitel 9 Fehler vermeiden 335
9.1 Einführung 335
9.2 Testen und Fehlerbeseitigung 336
9.3 Modultests in BlueJ 337
9.4 Tests automatisieren 343
9.5 Refactoring bei Streams einsetzen (fortgeschritten) 351
9.6 Debugging 351
9.7 Kommentierung und Programmierstil 353
9.8 Manuelle Ausführung 354
9.9 Ausgabeanweisungen 360
9.10 Debugger 363
9.11 Das Debugging von Streams (fortgeschritten) 365
9.12 Die Wahl der richtigen Teststrategie 366
9.13 Techniken umsetzen 367

Teil II Anwendungsstrukturen 369

Kapitel 10 Bessere Struktur durch Vererbung 371
10.1 Das Beispiel „Netzwerk“ 372
10.2 Einsatz von Vererbung 382
10.3 Vererbungshierarchien 384
10.4 Vererbung in Java 385
10.5 Weitere Einsendungen für Netzwerk 389
10.6 Vorteile durch Vererbung (bis hierher) 391

10.7 Subtyping 391
10.8 Die Klasse Object 397
10.9 Die Hierarchie der Sammlungstypen 398

Kapitel 11 Mehr über Vererbung 403
11.1 Das Problem: die Methode zum Anzeigen 403
11.2 Statischer und dynamischer Typ 405
11.3 Überschreiben von Methoden 408
11.4 Dynamische Methodensuche 410
11.5 super-Aufrufe in Methoden 413
11.6 Methoden-Polymorphie 414
11.7 Methoden aus Object: toString 414
11.8 Objektgleichheit: equals und hashCode 417
11.9 Der Zugriff über protected 420
11.10 Der Operator instanceof 422
11.11 Ein weiteres Beispiel für Vererbung mit Überschreiben 423

Kapitel 12 Weitere Techniken zur Abstraktion 429
12.1 Simulationen 429
12.2 Die Fuechse-und-Hasen-Simulation 431
12.3 Abstrakte Klassen 444
12.4 Weitere abstrakte Methoden 451
12.5 Multiple Vererbung 453
12.6 Interfaces 457
12.7 Ein weiteres Beispiel für ein Interface 465
12.8 Die Klasse Class 467
12.9 Abstrakte Klasse oder Interface? 468
12.10 Ereignisgesteuerte Simulationen 468
12.11 Zusammenfassung der Vererbung 470

Kapitel 13 Grafische Benutzungsoberflächen 475
13.1 Einführung 475
13.2 Komponenten, Layout und Ereignisbehandlung 476
13.3 AWT und Swing 477
13.4 Das Beispiel: ein Bildbetrachter 478
13.5 Bildbetrachter 1.0: die erste komplette Version 490
13.6 Bildbetrachter 2.0: die Programmstruktur verbessern 504
13.7 Bildbetrachter 3.0: weitere GUI-Komponenten 509
13.8 Innere Klassen 513
13.9 Zusätzliche Erweiterungen 518
13.10 Ein weiteres Beispiel: der Musikplayer 520

Kapitel 14 Fehlerbehandlung 527
14.1 Das Adressbuch-Projekt 528
14.2 Defensive Programmierung 531
14.3 Fehlermeldungen durch den Dienstleister 534
14.4 Prinzipien der Exception-Behandlung 539
14.5 Die Behandlung von Exceptions 545
14.6 Neue Exception-Klassen definieren 552
14.7 Die Verwendung von Zusicherungen 553
14.8 Wiederaufsetzen und Fehlervermeidung 557
14.9 Dateibasierte Ein- und Ausgabe 560

Kapitel 15 Entwurf von Anwendungen 573
15.1 Analyse und Entwurf 573
15.2 Klassenentwurf 581
15.3 Dokumentation 583
15.4 Kooperation 584
15.5 Prototyping 585
15.6 Softwarewachstum 586
15.7 Der Einsatz von Entwurfsmustern 588

Kapitel 16 Eine Fallstudie 597
16.1 Die Fallstudie 597
16.2 Analyse und Entwurf 599
16.3 Klassenentwurf 602
16.4 Iterative Entwicklung 608
16.5 Ein weiteres Beispiel 616
16.6 Ein Blick nach vorn 616

Anhang A Arbeiten mit BlueJ-Projekten 617
A.1 BlueJ installieren 617
A.2 Ein Projekt öffnen 617
A.3 Der Debugger in BlueJ 617
A.4 BlueJ konfigurieren 618
A.5 Auf deutsche Schnittstelle umstellen 618
A.6 Einbinden einer lokalen API-Dokumentation 619
A.7 Vorlagen für neue Klassen ändern 619

Anhang B Datentypen in Java 621
B.1 Primitive Typen 621
B.2 Cast-Operator für primitive Typen 622
B.3 Objekttypen 623

B.4 Wrapper-Klassen 623
B.5 Cast-Operator für Objekttypen 624

Anhang C Operatoren 625
C.1 Arithmetische Ausdrücke 625
C.2 Boolesche Ausdrücke 626
C.3 Abkürzungsoperatoren 627

Anhang D Kontrollstrukturen in Java 629
D.1 Kontrollstrukturen 629
D.2 Auswahlanweisungen 629
D.3 Schleifen 632
D.4 Exceptions 634
D.5 Zusicherungen 635

Anhang E Java ohne BlueJ 637
E.1 Java ohne BlueJ ausführen 637
E.2 Konsolenanwendungen und die Problematik der Umlaute 639
E.3 Ausführbare jar-Dateien erzeugen 641
E.4 Entwickeln ohne BlueJ 642

Anhang F Benutzung des Debuggers 643
F.1 Haltepunkte 644
F.2 Die Kontrollknöpfe 644
F.3 Anzeige der Variablen 645
F.4 Die Anzeige der Aufruffolge 646
F.5 Die Thread-Anzeige 646

Anhang G Testwerkzeuge für Modultests mit JUnit 647
G.1 Aktivieren der Test-Funktionalität 647
G.2 Eine Testklasse erzeugen 647
G.3 Eine Testmethode erzeugen 647
G.4 Zusicherungen bei Tests 648
G.5 Tests ausführen 648
G.6 Testgerüste 648

Anhang H Werkzeuge für die Teamarbeit 649
H.1 Server-Einrichtung 649
H.2 Teamarbeit-Funktionalität aktivieren 649
H.3 Ein Projekt zur gemeinsamen Nutzung einrichten 649
H.4 An einem Projekt gemeinsam arbeiten 650
H.5 Aktualisieren und Abgeben 650
H.6 Weitere Informationen 650

Anhang I Javadoc 651
I.1 Dokumentationskommentare 651
I.2 Unterstützung für Javadoc in BlueJ 654

Anhang J Quelltextkonventionen 655
J.1 Benennung 655
J.2 Layout 656
J.3 Dokumentation 657
J.4 Restriktionen bei der Sprachbenutzung 658
J.5 Programmiermuster 659

Anhang K Wichtige Bibliotheksklassen 661
K.1 Das Paket java.lang 661
K.2 Das Paket java.util 662
K.3 Die Pakete java.io und java.nio.file 664
K.4 Das Paket java.util.function 665
K.5 Das Paket java.net 666
K.6 Weitere wichtige Pakete 666

Register 667

Widmung

Für meine Frau Helen

–djb

Für K.W.

–mk

Vorwort von James Gosling, Erfinder von Java

Es war eine schmerzhafte Erfahrung für mich, meine Tochter Kate und die anderen Kinder in ihrer Mittelstufenklasse mit einem Java-Kurs kämpfen zu sehen, in dem eine kommerzielle Entwicklungsumgebung benutzt wurde. Die Leistungsfähigkeit des Werkzeugs hat die Komplexität des Lernprozesses erheblich erhöht. Ich wünschte, ich hätte früher begriffen, was da passierte. Leider konnte ich mit dem Lehrer erst über das Problem sprechen, als es schon zu spät war. Dies ist eine Situation, für die BlueJ ideal geeignet ist.

BlueJ ist eine interaktive Entwicklungsumgebung mit einem Auftrag: Sie wurde speziell für Studenten entwickelt, die Programmieren lernen. Sie wurde entworfen von Dozenten, die seit vielen Jahren Tag für Tag Objektorientierung lehren. Es war erfrischend, mit den Entwicklern von BlueJ zu sprechen: Sie haben eine sehr klare Vorstellung von ihrem Ziel. Unsere Diskussionen handelten meist eher davon, welche Dinge weggelassen werden sollten, und weniger davon, welche noch hinzugenommen werden sollten. BlueJ ist sehr klar strukturiert und ausgesprochen zielgerichtet.

Aber dies ist kein Buch über BlueJ. Es geht ums Programmieren.

In Java.

In den letzten Jahren hat Java in der Programmierausbildung eine immer größere Bedeutung bekommen. Dafür gibt es mehrere Gründe. Einer ist, dass Java viele Eigenschaften besitzt, die die Sprache für die Lehre geeignet machen: eine relativ klare Definition; ausführliche Prüfungen zur Übersetzungszeit durch den Compiler, die den Studenten frühes Feedback bei Problemen geben; und ein sehr robustes Speichermodell, das viele „mysteriöse“ Fehler von vornherein ausschaltet, die durch das Überschreiten von Objektgrenzen oder Typregeln auftreten können. Ein weiterer Grund ist, dass Java inzwischen kommerziell sehr interessant geworden ist.

Dieses Buch geht direkt das am schwierigsten zu vermittelnde Konzept an: Objekte. Das Buch führt die Studenten von den allerersten Schritten hin zu sehr anspruchsvollen Konzepten.

Und es löst eines der schwierigsten Probleme für Lehrbücher über Programmierung: Es thematisiert explizit auch das Schreiben und Ausführen von Programmen. Die meisten Lehrbücher gehen über diese Punkte stillschweigend hinweg oder streifen

sie nur kurz und überlassen es dem Dozenten, diese Aspekte zu klären. Dieser wird mit dem Problem alleingelassen, das zu vermittelnde Material in Beziehung zu setzen zu den Schritten, die zur Lösung der Übungen gemacht werden müssen. Stattdessen setzt dieses Buch den Einsatz von BlueJ voraus und kann deshalb das Verstehen der Konzepte auf die Mechanismen, die ihre Erforschung ermöglichen, abstimmen.

Ich wünschte, für meine Tochter hätte BlueJ schon im letzten Jahr zur Verfügung gestanden. Vielleicht im nächsten Jahr …

Vorwort für den Lehrenden

Neues in der sechsten Auflage

Dies ist die sechste Auflage dieses Buchs und der Inhalt wurde – wie immer bei Neuauflagen – an die jüngsten Entwicklungen bei objektorientierten Programmen angepasst.

Viele der Änderungen können dieses Mal oberflächlich einer neuen Java-Version zugeschrieben werden: Java 8. Diese Version wurde 2014 herausgegeben und wird jetzt sehr häufig in der Praxis eingesetzt. In der Tat wurde bisher keine Java-Version so schnell angenommen – daher ist es jetzt an der Zeit, auch die Art zu verändern, wie wir Einsteiger darin unterrichten.

Die Veränderungen umfassen jedoch mehr als nur das Hinzufügen von einigen neuen Sprachkonstrukten. Die wichtigsten neuen Aspekte von Java 8 drehen sich um neue Konstrukte, die einen (teilweise) funktionalen Programmierstil unterstützen. Die wachsende Beliebtheit der funktionalen Programmierung hat diese Änderung vorangetrieben. Der Unterschied ist viel tiefer und fundamentaler und geht über das bloße Hinzufügen von neuer Syntax hinaus. Und es ist das Wiedererwachen von funktionalen Ideen in der Programmierung allgemein – nicht nur die Existenz von Java 8 –, was es nahelegt, diese Aspekte jetzt in einer modernen Auflage eines Programmierlehrbuchs zu behandeln.

Die Ideen und Techniken der funktionalen Programmierung – obwohl recht alt und im Prinzip gut bekannt – haben in den letzten Jahren einen deutlichen Popularitätsschub erlebt. Es wurden neue Sprachen entwickelt und ausgewählte funktionale Techniken in die bestehenden traditionellen imperativen eingebettet. Einer der Hauptgründe dafür sind die Neuerungen bei der verfügbaren Computerhardware sowie ein Wandel der Problemstellungen, die wir angehen wollen.

Fast alle Programmierplattformen bieten heutzutage parallele Verarbeitung. Selbst Laptops der Mittelklasse und Mobiltelfefone haben Prozessoren mit mehreren Kernen, wodurch Parallelität eine echte Möglichkeit auf alltäglich genutzten Geräten ist. Doch in der Praxis setzt sich dieser Trend noch nicht im größeren Rahmen fort.

Anwendungen zu entwickeln, die eine optimale Ausnutzung von paralleler Verarbeitung und von mehreren Prozessoren bieten, ist sehr, sehr schwierig. Die wenigsten der heute verfügbaren Anwendungen nutzen die aktuelle Hardware in dem Ausmaß, wie es theoretisch möglich wäre.

Dies wird sich auch nicht viel ändern: Die Möglichkeit (und Herausforderung) von paralleler Hardware wird bestehen bleiben, und die Programmierung dieser Geräte mit traditionellen imperativen Sprachen wird nicht einfacher werden.

An dieser Stelle kommt die funktionale Programmierung ins Spiel.

Mit funktionalen Sprachkonstrukten ist es möglich, ein wenig Parallelität sehr effizient zu automatisieren. Programme können potenziell mehrere Kerne ohne viel Aufwand seitens des Programmieres verwenden. Funktionale Konstrukte haben andere Vorteile – eleganterer Ausdruck für bestimmte Probleme und häufig eine bessere Lesbarkeit –, aber es ist die Fähigkeit, mit Parallelität umzugehen, die sicherstellen wird, dass funktionale Programmieraspekte uns noch lange begleiten werden.

Jeder Lehrer, der seine Schüler auf die Zukunft vorbereiten möchte, sollte ihnen auch ein gewisses Verständnis für funktionale Aspekte vermitteln. Ohne diese Kenntnisse wird man es nicht mehr zur Meisterschaft im Programmieren bringen können. Ein Neuling muss sicher nicht die gesamte funktionale Programmierung beherrschen, aber ein Grundverständnis davon – und was wir damit erreichen können – wird bald wesentlich sein.

Wann genau funktionale Techniken eingeführt werden sollten, ist eine interessante Frage. Wir glauben nicht, dass es eine einzige richtige Antwort darauf gibt; verschiedene Abfolgen sind denkbar. Funktionale Programmierung könnte als fortgeschrittenes Thema am Ende des gesamten Buchs behandelt werden oder man könnte es ansprechen, wenn wir zum ersten Mal auf die Bereiche treffen, für die es anwendbar ist, als Alternative zu den imperativen Techniken. Man könnte das Thema sogar zuerst behandeln.

Eine weitere Frage ist, wie mit dem traditionellen Programmierstil in den Bereichen umgegangen werden soll, in denen die funktionalen Konstrukte nun verfügbar sind: Sollten sie ersetzt werden oder behandelt man beide?

Wir sind für dieses Buch davon ausgegangen, dass jeder Lehrer seine eigenen Bedürfnisse und Vorlieben hat. Deshalb haben wir eine Struktur entwickelt, die – so hoffen wir – unterschiedliche Ansätze ermöglicht, je nach den Vorzügen des Lernenden oder Lehrers.

- Wir haben die „alten“ Techniken nicht ersetzt. Wir behandeln den neuen funktionalen Ansatz zusätzlich zu den bestehenden Inhalten. Funktionale Konstrukte in Java sind am herausragendsten, wenn man es mit Sammlungen von Objekten zu tun hat, und die traditionellen Konzepte – die Verwendung von Schleifen und expliziter Iteration – zu beherrschen, ist für jeden Programmierer immer noch wesentlich. Nicht nur, weil es Millionen Codezeilen in der Welt gibt, die in diesem Stil geschrieben sind – und weiterhin geschrieben werden –, sondern es gibt auch spezielle Fälle, in denen diese Techniken genutzt werden müssen, selbst wenn man generell die neuen funktionalen Konstrukte vorzieht. Das Ziel ist es, beide Techniken zu beherrschen.
- Wir präsentieren das neue funktional-orientierte Material im Buch an den Stellen, an denen wir die Probleme besprechen, die mit diesen Konstrukten angegangen werden. Zum Beispiel behandeln wir die funktionale Verarbeitung von Sammlungen, sobald wir den Sammlungen begegnen.

- Kapitel und Abschnitte, die dieses neue Material behandeln, sind jedoch deutlich als „fortgeschritten" markiert und so strukturiert, dass sie gefahrlos beim ersten Lesen übersprungen (oder überhaupt ausgelassen) werden können.
- Die beiden vorigen Punkte ermöglichen unterschiedliche Herangehensweisen, dieses Buch durchzuarbeiten: Wenn es die Zeit erlaubt, kann das Buch in der vorgestellten Reifenfolge gelesen werden, damit wird der gesamte Inhalt abgedeckt, einschließlich der funktionalen Ansätze als Alternative zu den imperativen – wenn die Probleme auftauchen, die wir mit funktionalen Konzepten lösen wollen. Ist die Zeit jedoch knapp, können diese weiterführenden Abschnitte übersprungen werden, die Betonung wird dann erst einmal auf ein gründliches Durchdringen der imperativen, objektorientierten Programmierung gesetzt. (Wir sollten betonen, dass *funktional* nicht der Gegensatz von *objektorientiert* ist: Egal, ob das funktionale Material in das Studium einbezogen wird oder ob das Augenmerk im Großen und Ganzen auf den imperativen Techniken liegt – jeder Leser dieses Buchs wird am Ende mit einem guten Verständnis der Objektorientierung ausgestattet sein!) Eine weitere Möglichkeit des Zugangs ist es, die weiterführenden Abschnitte erst einmal zu überspringen und diese später als eine separate Einheit zu behandeln. Sie enthalten alternative Herangehensweisen zu anderen Konstrukten und können unabhängig bearbeitet werden.

Wir hoffen, dies macht deutlich, dass dieses Buch Flexibilität bietet, falls es gewünscht ist, aber auch Orientierungshilfe für Leser, die noch keine klare Vorliebe haben: Dann kann man das Buch einfach von vorne nach hinten lesen.

Abgesehen von den bisher genannten größeren Änderungen, präsentiert diese Auflage auch zahlreiche kleinere Verbesserungen. Die Gesamtstruktur, der Ton und die Herangehensweise sind unverändert: Sie haben sich in der Vergangenheit bewährt und es gibt keinen Grund, davon abzurücken. Doch wir überarbeiten das Material ständig und suchen nach Verbesserungsmöglichkeiten. Wir haben jetzt fast 15 Jahre kontinuierliche Lehrerfahrung mit diesem Buch und dies schlägt sich überall in den vielen kleineren Verbesserungen nieder.

Dieses Buch ist eine Einführung in die objektorientierte Programmierung für Programmieranfänger. Der Fokus dieses Buches liegt auf allgemeinen objektorientierten Programmierkonzepten aus der Sicht der Softwaretechnik.

Während die ersten Kapitel auf Studenten ohne jegliche Programmiererfahrung abzielen, sind spätere Kapitel auch für fortgeschrittene oder professionelle Programmierer interessant. Insbesondere sollten auch Programmierer von diesem Buch profitieren können, die Erfahrung in einer nicht objektorientierten Sprache haben und ihre Fähigkeiten in Richtung Objektorientierung ausbauen möchten.

Wir verwenden das gesamte Buch hindurch zwei Werkzeuge, um die vorgestellten Konzepte in die Praxis umzusetzen: die Programmiersprache Java und die Java-Entwicklungsumgebung BlueJ.

Java

Die Sprache Java wurde wegen ihres Sprachentwurfs und ihrer Popularität gewählt. Die Programmiersprache Java selbst bietet eine saubere Umsetzung der meisten wichtigen objektorientierten Konzepte und eignet sich gut als einführende Sprache. Ihre Popularität garantiert einen enormen Vorrat an unterstützendem Material.

Auf jedem Lehrgebiet ist eine große Menge unterschiedlicher Quellen sowohl für Lehrende als auch für Studenten hilfreich. Für Java gibt es zahllose Bücher, Beschreibungen, Übungen, Compiler, Entwicklungsumgebungen und Fragenkataloge in den unterschiedlichsten Arten und Stilen. Viele sind online verfügbar und viele sind kostenlos. Die große Menge und die gute Qualität an Zusatzmaterial machen Java zu einer exzellenten Wahl als Einführung in die objektorientierte Programmierung.

Wenn so viel Java-Material bereits verfügbar ist, gibt es dann überhaupt noch etwas Neues dazu zu sagen? Wir denken ja, und das zweite Werkzeug ist einer der Gründe dafür …

BlueJ

BlueJ erfordert einige Erläuterungen. Dieses Buch ist einzigartig in seiner kompletten Integration der BlueJ-Umgebung.

BlueJ ist eine Java-Entwicklungsumgebung, die an der Deakin-Universität, Australien, und der Universität von Kent in Canterbury, England, ausdrücklich mit dem Ziel entwickelt wird, für die Einführung in die objektorientierte Programmierung zu dienen. Sie eignet sich aus mehreren Gründen für diese Lehrsituation besser als andere Umgebungen:

- Die Benutzungsschnittstelle ist sehr viel einfacher. Programmieranfänger können die BlueJ-Umgebung typischerweise bereits nach 20 Minuten Einführung kompetent benutzen. Von diesem Zeitpunkt an kann sich die Darstellung auf die wichtigen Konzepte konzentrieren – Objektorientierung und Java –, statt wertvolle Zeit mit der Diskussion über Umgebungen, Dateisysteme, Klassenpfade, DOS-Kommandos oder DLL-Konflikte zu vergeuden.
- BlueJ bietet wichtige Lehrwerkzeuge, die in anderen Umgebungen nicht zur Verfügung stehen. Eines davon ist die Visualisierung der Klassenstruktur. BlueJ zeigt automatisch ein UML-ähnliches Diagramm der Klassen und Beziehungen in einem Projekt. Die Visualisierung dieser wichtigen Konzepte ist eine große Hilfe für Lehrende und Studierende. Es ist nicht leicht, das Konzept von Objekten zu verstehen, wenn auf dem Bildschirm ausschließlich Programmtext zu sehen ist! Die Notation der Diagramme ist eine einfache Untermenge der UML, zugeschnitten auf den Bedarf von Programmieranfängern. Dies macht es leicht verständlich, erlaubt aber gleichzeitig die Migration auf die volle UML in späteren Kursen.
- Eine der wichtigsten Stärken der BlueJ-Umgebung ist die Möglichkeit für den Benutzer, direkt Objekte von beliebigen Klassen erzeugen und die Methoden dieser Objekte aufrufen zu können. Dies gibt die Möglichkeit zum unmittelbaren Umgang mit Objekten ohne großen Aufwand durch die Umgebung. Studenten bekommen förmlich ein „Gefühl" dafür, wie Objekte erzeugt, Methoden aufgerufen, Parameter übergeben und Ergebnisse zurückgeliefert werden. Sie können eine Methode ausprobieren, unmittelbar nachdem sie sie geschrieben haben, ohne Testklassen schreiben zu müssen. Diese Möglichkeit ist eine unschätzbare Hilfe beim Verstehen der zugrunde liegenden Konzepte und Sprachdetails.
- BlueJ weist zahlreiche weitere Tools und Charakteristika auf, die speziell für Studenten der Softwareentwicklung entworfen wurden. Einige sind dafür gedacht, grundlegende Konzepte besser zu verstehen (wie z.B. die Hervorhebung von Sichtbarkeitsbereichen), andere dienen dazu, die Studenten, sobald sie bereit

dafür sind, an fortgeschrittene Werkzeuge und Techniken heranzuführen, wie z.B. das Testen mit JUnit oder die Teamarbeit mit einem Versionsüberwachungsssystems wie Subversion. Einige dieser Features gibt es nur in der BlueJ-Umgebung.

BlueJ ist eine vollständige Java-Entwicklungsumgebung. Es ist keine zurechtgestutzte, vereinfachte Version von Java für die Lehre. Sie läuft auf dem Java Development Kit von Oracle und benutzt den Standardcompiler und die zugehörige virtuelle Maschine. Dies garantiert, dass sie immer mit der offiziellen und aktuellen Java-Spezifikation übereinstimmt.

Die Autoren dieses Buches haben mehrere Jahre Lehrerfahrung mit der BlueJ-Umgebung (und viele Jahre mehr aus der Zeit vor BlueJ). Wir haben beide festgestellt, dass die Verwendung von BlueJ bei den Studenten in unseren Kursen zu größerem Engagement, besserem Verständnis und gesteigerter Aktivität geführt hat. Einer der Autoren dieses Buches ist außerdem der Leiter der Entwicklung des BlueJ-Systems.

Objekte wirklich zuerst

Unsere Wahl ist unter anderem deshalb auf BlueJ gefallen, weil mit dieser Umgebung wirklich die wichtigen Konzepte zuerst vorgestellt werden können. „Objects First" war lange Zeit der Schlachtruf vieler Buchautoren und Lehrer. Leider macht die Sprache Java die Umsetzung dieses hehren Ansatzes nicht leicht. Zahlreiche Hürden in Syntax und anderen Details müssen überwunden werden, bevor die ersten „lebenden" Objekte sichtbar werden. Der minimale Ablauf zum Erzeugen und Aufrufen eines Objekts in Java schließt typischerweise folgende Schritte ein:

- das Schreiben einer Klasse
- das Schreiben einer `main`-Methode unter Benutzung der Konzepte von statischen Methoden, Parametern und Arrays in der Signatur
- eine Anweisung zur Objekterzeugung (mit `new`)
- eine Zuweisung an eine Variable
- eine Variablendeklaration, inklusive Typ
- einen Methodenaufruf mit der Punkt-Notation
- möglicherweise eine Parameterliste

Aus diesem Grund müssen die meisten Lehrbücher entweder

- diese Liste zuerst durcharbeiten und können deshalb erst einige Kapitel später auf Objekte zu sprechen kommen oder
- ein Programm im Stil von „Hello, World" mit einer einzelnen statischen Methode als erstes Beispiel wählen und damit gar keine Objekte erzeugen.

Mit BlueJ ist das kein Problem. Ein Student kann als erste Aktivität ein Objekt erzeugen und seine Methoden aufrufen! Weil Benutzer Objekte direkt erzeugen und manipulieren können, können Konzepte wie Klassen, Objekte, Methoden und Parameter ohne Weiteres direkt diskutiert werden, bevor die erste Zeile Java-Quelltext betrachtet werden muss. Statt an dieser Stelle weitere Details zu beschreiben, empfehlen wir dem neugierigen Leser einen schnellen Blick in Kapitel 1 – dort werden die hier gemachten Aussagen schnell deutlich.

Iteratives Vorgehen

Ein weiterer wichtiger Aspekt an diesem Buch ist das iterative Vorgehen. Ein in Kreisen der Informatikpädagogik bekanntes Lehrmuster (aus dem „pedagogical patterns project") besagt, dass wichtige Konzepte früh und häufig geschult werden sollten.[1] Es ist eine große Versuchung für den Autor eines Lehrbuches, bei der Vorstellung eines Konzepts alle seine Aspekte darzustellen. Beispielsweise ist es üblich, bei der Einführung des Typkonzepts eine vollständige Liste der vordefinierten Datentypen anzugeben oder bei der Vorstellung von Schleifen alle Schleifenarten zu diskutieren.

Diese beiden Ansätze stehen im Konflikt miteinander: Wir können uns nicht einerseits auf die wichtigen Konzepte konzentrieren und gleichzeitig Vollständigkeit fordern. Unsere Erfahrung mit Lehrbüchern ist, dass durch zu viele Details zu Anfang die wichtigen Konzepte untergehen und damit schwerer erfassbar werden.

In diesem Buch werden wir wichtige Punkte immer wieder ansprechen, sowohl innerhalb eines Kapitels als auch über die Kapitel hinweg. Konzepte werden üblicherweise so eingeführt, dass sie das vorliegende Problem lösen. Sie werden dann später aus einem anderen Blickwinkel erneut betrachtet, sodass das Verständnis in den späteren Kapiteln immer weiter vertieft wird. Dieser Ansatz erleichtert auch ein Verständnis der Konzepte, die gegenseitig voneinander abhängen.

Einige Dozenten sind möglicherweise mit dem iterativen Vorgehen nicht vertraut. Beim Blick auf die ersten Kapitel könnten Dozenten, die mit einer eher sequenziellen Vorgehensweise vertraut sind, über die Fülle der angesprochenen Konzepte überrascht sein. Dies scheint eine sehr steile Lernkurve zu implizieren.

Aber die Geschichte endet nicht an dieser Stelle. Von den Studenten wird nicht erwartet, dass sie alle Details dieser Konzepte unmittelbar verstehen. Stattdessen werden die fundamentalen Konzepte im Laufe des Buches immer wieder angesprochen, sodass die Studenten ein immer besseres Verständnis bekommen. Da sich ihr Wissensstand während der Beschäftigung mit diesen Themen stetig verändert, können sie durch die Wiederholung wichtiger Konzepte ein tieferes Verständnis erlangen.

Wir haben dieses Vorgehen viele Male mit Studenten erprobt. Manchmal haben die Studenten mit diesem Vorgehen weniger Probleme als die Lehrenden. Und bedenken Sie: Eine steile Lernkurve ist nicht problematisch, solange Sie sicherstellen, dass Ihre Studenten den Anstieg schaffen können!

Keine komplette Abdeckung der Sprache Java

Verbunden mit unserem iterativen Vorgehen ist die Entscheidung, dass wir nicht versuchen wollen, die Sprache Java in diesem Buch komplett zu behandeln.

Der Fokus dieses Buches liegt auf der Vermittlung allgemeiner objektorientierter Programmierprinzipien, nicht von Details der Sprache Java. Studenten, die mit

1 Das „Early Bird"-Muster, in J. Bergin: „Fourteen pedagogical patterns for teaching computer science", Proceedings of the *Fifth European Conference on Pattern Languages of Programs* (EuroPLoP 2000), Irsee, Deutschland, Juli 2000.

diesem Buch Programmieren lernen, können leicht 30 bis 40 Jahre als professionelle Softwareentwickler arbeiten – da ist es eine ziemlich sichere Wette, dass der Großteil ihrer Arbeit nicht in Java stattfinden wird. Natürlich sollte jedes Lehrbuch den Anspruch haben, mehr Fundamentales zu vermitteln als nur die Programmiersprache, die gerade in Mode ist.

Auf der anderen Seite sind viele Details von Java wichtig, um die praktischen Aufgaben erledigen zu können. In diesem Buch besprechen wir die Java-Konstrukte so detailliert wie notwendig, um die Konzepte verstehen und die praktischen Aufgaben lösen zu können. Einige Java-spezifische Konstrukte hingegen wurden bewusst ausgelassen.

Uns ist klar, dass einige Dozenten Themen behandeln werden, die wir nicht im Detail besprechen. Statt zu versuchen, alle Themen selbst abzudecken (und damit den Umfang dieses Buches auf 1500 Seiten aufzublähen), versuchen wir „Aufhänger“ anzubieten. Solche Aufhänger sind Verweise, häufig in Form einer Frage, die ein Thema anspricht und einen Hinweis auf einen Anhang oder externes Material gibt. So wird sichergestellt, dass bestimmte Themen zum passenden Zeitpunkt angesprochen werden, und es bleibt dem Leser oder dem Lehrenden überlassen, wie weit dem Verweis gefolgt werden sollte. Die Aufhänger dienen als Erinnerung an ein Thema und als Platzhalter, an denen eine vertiefende Diskussion vorgenommen werden kann.

Lehrende können dem von uns vorgegebenen Pfad im Buch folgen oder an entsprechenden Stellen in Seitenpfade verzweigen.

In den Kapiteln sind häufig auch Fragen enthalten, die verwandte Themen ansprechen, ohne dass diese im Buch diskutiert werden. Wir empfehlen Lehrenden, einige dieser Fragen im Kurs zu besprechen, bzw. den Studierenden, diese Übungsaufgaben zu bearbeiten.

Projektorientiertes Vorgehen

Die Einführung von neuen Themen in diesem Buch orientiert sich an Projekten. Das Buch diskutiert zahlreiche Programmierprojekte und bietet viele Übungen. Statt ein neues Konzept vorzustellen und es dann anhand einer Übungsaufgabe zur Lösung anwenden zu lassen, stellen wir zuerst ein Ziel auf und formulieren eine Aufgabe. Beim Analysieren der Aufgabe stellen wir fest, welche Lösungen wir benötigen. Als Konsequenz werden neue Konstrukte dann vorgestellt, wenn sie uns bei der Lösung einer Aufgabenstellung unterstützen können.

Die ersten Kapitel enthalten mindestens zwei Diskussionsbeispiele. Das sind Projekte, die detailliert besprochen werden, um die wichtigen Konzepte in jedem Kapitel zu illustrieren. Die Verwendung zweier unterschiedlicher Beispiele unterstützt das iterative Vorgehen: Jedes Konzept wird nach seiner Einführung noch einmal in einem anderen Kontext wiederholt.

Beim Entwurf dieses Buches haben wir versucht, viele unterschiedliche Beispielprojekte zu finden. Dies weckt hoffentlich die Neugierde des Lesers und illustriert auch die Bandbreite der Bereiche, in denen die Konzepte angewendet werden können. Wir hoffen, dass unsere Projekte den Lehrenden gute Startpunkte liefern und viele Ideen für vielfältige und spannende Aufgaben bieten.

Die Implementierung aller Projekte wurde sehr sorgfältig vorgenommen, sodass beim Lesen des Quelltextes auch viele Nebenaspekte betrachtet werden können. Wir glauben sehr stark an das Lernen durch Lesen und Nachahmen guter Beispiele. Damit dies funktioniert, ist es wichtig, dass die Beispiele gut und nachahmenswert geschrieben sind. Wir haben versucht, interessante Beispiele zu finden.

Alle Projekte sind als offene Problemstellungen entworfen. Auch wenn eine oder mehrere Versionen jedes Problems ausführlich im Buch dargestellt werden, sind alle Projekte so entworfen, dass zusätzliche Erweiterungen und Verbesserungen in Studentenprojekten vorgenommen werden können. Der komplette Quelltext aller Projekte ist beigefügt. Eine Liste der Projekte, die in diesem Buch diskutiert werden, ist ab Seite 26 zu finden.

Konzepte statt Sprachkonstrukte

Ein weiterer Aspekt, der dieses Buch von vielen anderen unterscheidet, ist die Orientierung an fundamentalen Aufgaben der Softwareentwicklung und nicht notwendigerweise an spezifischen Java-Konstrukten. Ein Hinweis darauf sind die Kapitelüberschriften. In diesem Buch werden Sie keine klassischen Kapitelüberschriften wie „Primitive Datentypen“ oder „Kontrollstrukturen“ finden. Die Strukturierung nach fundamentalen Entwicklungsaufgaben ermöglicht uns eine sehr viel allgemeinere Einführung, die sich nicht an den Besonderheiten einer speziellen Programmiersprache orientiert. Wir glauben auch, dass es den Studenten leichter fällt, der Motivation der jeweiligen Einleitung zu folgen, und dass die Lektüre dadurch interessanter wird.

Ein Resultat dieses Vorgehens ist, dass sich dieses Buch weniger als eine Referenz eignet. Einführende Lehrbücher und Referenzbücher haben unterschiedliche und teilweise konkurrierende Ziele. Bis zu einem gewissen Grad kann ein Lehrbuch versuchen, beides zu bieten, aber irgendwann muss ein Kompromiss eingegangen werden. Unser Buch ist deutlich als ein Lehrbuch entworfen, und immer, wenn es zu einem Konflikt kam, haben wir den Stil eines Lehrbuches dem eines Referenzbuches vorgezogen.

Dennoch haben wir Unterstützung für die Verwendung als Referenzbuch geboten, indem wir die Java-Konstrukte, die in den einzelnen Kapiteln angesprochen werden, in der Einführung jedes Kapitels aufgeführt haben.

Übersicht der Kapitel

Kapitel 1 beschäftigt sich mit den fundamentalen Konzepten der Objektorientierung: Objekte, Klassen und Methoden. Es liefert eine solide, pragmatische Einführung in diese Konzepte, ohne auf Details der Java-Syntax einzugehen. Nebenbei wird auch bereits das Konzept der Abstraktion kurz eingeführt – ein Thema, das sich durch viele Kapitel ziehen wird. Schließlich wird auch ein erster Blick auf den Quelltext geworfen. Zwei Projekte beschäftigen sich mit grafischen Figuren, die interaktiv gezeichnet werden können, ein weiteres Projekt behandelt das Anmelden von Studenten für Laborkurse.

Kapitel 2 öffnet Klassendefinitionen und untersucht, wie mit Quelltext das Verhalten von Objekten definiert wird. Wir diskutieren, wie Datenfelder definiert und Methoden implementiert werden, und arbeiten heraus, welche wichtige Rolle der Konstruktor beim Aufbau des Objektszustands, verkörpert durch seine

Felder, spielt. Hier führen wir auch die ersten Anweisungen ein. Das Hauptbeispiel ist die Implementierung eines Ticketautomaten. Wir untersuchen das Beispiel mit den Laborkursen aus Kapitel 1 ein bisschen genauer.

Kapitel 3 geht einen Schritt weiter, indem die Interaktion zwischen mehreren Objekten diskutiert wird. Wir sehen, wie Objekte zusammenarbeiten können, indem sie gegenseitig ihre Methoden aufrufen, um eine gemeinsame Aufgabe zu lösen. Wir diskutieren auch, wie ein Objekt weitere Objekte erzeugen kann. Die Anzeige einer digitalen Uhr wird diskutiert, die zwei Objekte zur Anzeige von Stunden und Minuten benutzt. Eine Version des Projekts, die eine grafische Benutzungsoberfläche umfasst, verfolgt eine Strategie, die uns im weiteren Verlauf des Buches immer wieder begegnen wird: Wir bieten für den interessierten und eifrigen Leser zusätzlichen Code, auf den wir jedoch im Text nicht ausführlich eingehen. Als zweites größeres Beispiel untersuchen wir die Simulation eines E-Mail-Systems, in dem Nachrichten zwischen E-Mail-Clients verschickt werden können.

In **Kapitel 4** fahren wir fort mit der Konstruktion größerer Objektstrukturen und greifen die Themen Abstraktion und Objektinteraktion noch einmal auf. Vor allem beginnen wir hier, Sammlungen von Objekten zu benutzen. Wir betrachten ein Verwaltungssystem für Musikdateien und ein Auktionssystem, um Sammlungen einzuführen. Gleichzeitig diskutieren wir das Iterieren über Sammlungen und werfen einen ersten Blick auf `for-each`- und `while`-Schleifen. Die erste Sammlung, die benutzt wird, ist die `ArrayList`.

In **Kapitel 5** beginnen wir, die fortgeschrittenen Konzepten vorzustellen (dieser Abschnitt kann übersprungen werden, wenn die Zeit knapp ist): Es ist eine Einführung in funktionale Programmierkonstrukte. Die funktionalen Konstrukte bieten eine Alternative zur imperativen Verarbeitung von Sammlungen, die wir in Kapitel 4 besprechen. Die dargestellten Probleme könnten auch ohne diese Techniken gelöst werden, doch funktionale Konstrukte eröffnen elegantere Möglichkeiten, unsere Ziele zu erreichen. Dieses Kapitel führt den funktionalen Ansatz allgemein ein sowie einige der neuen Sprachkonstrukte in Java.

Kapitel 6 behandelt Bibliotheken und Schnittstellen. Wir stellen die Standardbibliothek von Java vor und diskutieren einige wichtige Bibliotheksklassen. Am wichtigsten ist hier die Darstellung, wie die Dokumentation der Bibliothek gelesen und verstanden werden kann. Es wird diskutiert, wie wichtig das Schreiben von Dokumentationen für Softwareentwicklungsprojekte ist, und schließlich üben wir das Erstellen einer angemessenen Dokumentation für unsere eigenen Klassen. `Random`, `Set` und `Map` sind Beispiele von Klassen, die wir in diesem Kapitel untersuchen. Wir implementieren ein *Eliza*-ähnliches Dialogsystem und eine grafische Simulation von springenden Bällen, um diese Klassen anzuwenden.

Kapitel 7 konzentriert sich auf einen sehr speziellen Typ von Sammlung: Arrays. Das Arbeiten mit Arrays und der mit ihnen verbundenen Schleifenarten wird ausführlich besprochen.

In **Kapitel 8** diskutieren wir etwas formaler die Probleme beim Aufteilen eines Problems in geeignete Klassen für die Implementierung. Wir stellen Konzepte für guten Klassenentwurf vor, einschließlich Entwurf nach Verantwortlichkeiten, Kopplung, Kohäsion und Refactoring. Für diese Diskussion verwenden wir ein interakti-

ves, textbasiertes Adventure-Game (*Die Welt von Zuul*). Wir verbessern in mehreren Iterationen den internen Entwurf des Spiels und erweitern seine Funktionalität, bis wir schließlich mit einer umfangreichen Liste von Erweiterungsmöglichkeiten Anregungen für weiterführende Studentenprojekte geben.

Kapitel 9 beschäftigt sich mit Konzepten, mit denen korrekte, verständliche und wartbare Klassen geschrieben werden können. Es behandelt Aspekte, die vom Schreiben eines verständlichen Quelltextes (einschließlich Stil und Kommentierung) bis hin zum Testen (einschließlich formalisierter Regressions- und JUnit-Tests) und zur Fehlerbeseitigung reichen. Es werden Teststrategien vorgestellt und einige Methoden zur Fehlerbeseitigung ausführlich diskutiert. Wir benutzen das Beispiel eines Onlineshops und die Implementierung eines Taschenrechners für die Diskussion dieser Konzepte.

Kapitel 10 und **11** führen Vererbung, Polymorphie und damit verbundene Konzepte ein. Wir untersuchen einen Teil eines sozialen Netzwerks, um die Konzepte darzustellen. Fragen der Quelltextvererbung, des Subtypings, der polymorphen Methodenaufrufe und des Überschreibens von Methoden werden ausführlich diskutiert.

In **Kapitel 12** implementieren wir eine Jäger-Beute-Simulation. Sie dient als Ausgangspunkt für die Diskussion weiterer Abstraktionsmechanismen, die auf Vererbung aufsetzen, insbesondere Interfaces und abstrakte Klassen.

Kapitel 13 behandelt zwei neue Beispiele: einen Bildbetrachter und eine grafische Benutzungsoberfläche (GUI) für das Verwaltungssystem von Musiksammlungen (dem wir bereits in Kapitel 4 begegnet sind). Anhand dieser Beispiele wird dargestellt, wie GUIs konstruiert werden.

Kapitel 14 beschäftigt sich dann mit der schwierigen Frage, wie mit Fehlern zur Laufzeit umgegangen werden soll. Einige Probleme und Lösungen werden diskutiert und der Exception-Mechanismus von Java wird ausführlich vorgestellt. Wir erweitern und verbessern eine Adressbuch-Anwendung, um diese Konzepte darzustellen. Als Fallstudie für einen Aufgabenbereich, wo die Fehlerbehandlung unverzichtbar ist, dient uns die Ein-/Ausgabe.

Kapitel 15 beleuchtet ausführlicher die nächste Abstraktionsstufe: wie man zu einem vage beschriebenen Problem passende Klassen und Methoden findet. In den vorigen Kapiteln haben wir angenommen, dass große Teile der Anwendung bereits existieren und wir Verbesserungen vornehmen. Jetzt ist es an der Zeit, uns anzusehen, wie man mit einem Code bei Null beginnt. Dies schließt eine detaillierte Darstellung von Vorgehensweisen ein, wie wir die passenden Klassen für eine Anwendung finden, wie diese interagieren und wie die Zuständigkeiten verteilt sein sollten. Wir benutzen CRC-Karten als Herangehensweise, um ein Kinobuchungssystem zu entwerfen.

In **Kapitel 16** führen wir alles bis dahin Dargestellte zusammen. Es enthält eine komplette Fallstudie, die mit einem Anwendungsentwurf beginnt und über den Entwurf von Klassenschnittstellen und die Diskussion vieler wichtiger funktionaler und nichtfunktionaler Aspekte bis hin zu Details der Implementierung reicht. Konzepte aus früheren Kapiteln (wie Zuverlässigkeit, Datenstrukturen, Klassenentwurf, Testen und Erweiterbarkeit) werden erneut in einem neuen Kontext dargestellt.

Zusatzmaterial

Website zum Buch: Alle Projekte, die als Diskussionsbeispiele und Übungen in diesem Buch verwendet wurden, können von der Buchwebsite unter *http://www.bluej.org/objects-first/* heruntergeladen werden. Die Website enthält außerdem Links zum Download von BlueJ sowie weiteres Material.

Die Companion Website (CWS) für Studenten: Die folgenden Ressourcen stehen allen Lesern dieses Buches im Internet unter *http://www.pearson-studium.de/java-lernen-mit-bluej.html* zur Verfügung:[2]

- Quelltext zu allen Projekten
- Links zu weiterem interessanten Material

Auf der englischsprachigen Seite *http://bluej.org/objects-first/* finden Sie außerdem noch den Program Style Guide zu allen Beispielen in diesem Buch sowie Links zu allen im Buch erwähnten Ressourcen.

Diskussionsgruppen für Studenten: Studenten, die Fragen haben oder das Material in diesem Buch beziehungsweise Themen zu BlueJ allgemein diskutieren wollen, können dies unter *http://groups.google.com/group/bluej-discuss* in der Gruppe *bluej-discuss* machen.

Ressourcen für Dozenten: Das folgende Material steht nur qualifizierten Dozenten zur Verfügung:

- Lösungen zu den Übungen am Ende des Kapitels
- PowerPoint-Folien

Der Blueroom

Vielleicht noch wichtiger als das statische Material der oben genannten Websites ist das sehr aktive Forum der BlueJ-Gemeinde. Es wendet sich an Dozenten, die BlueJ und dieses Buch im Unterricht einsetzen. Das Forum läuft unter der Bezeichnung Blueroom und ist unter

http://blueroom.bluej.org

zu finden. Der Blueroom enthält eine Sammlung von umfangreichem Lehrmaterial, das von den Lehrkräften gemeinsam genutzt werden kann, sowie ein Diskussionsforum, in denen die Lehrkräfte Fragen stellen, Themen diskutieren und sich über die neuesten Entwicklung auf dem Laufenden halten können. Im Blueroom haben Sie die Möglichkeit, zu vielen anderen Dozenten, Entwicklern sowie den Autoren dieses Buches Kontakt aufzunehmen.

---

2 Oder geben Sie gerne in das Suchfeld unter www.pearson-studium.de die Buchnummer 4911 ein und gelangen direkt zur Webseite des Buches.

Danksagungen

Viele Menschen haben auf unterschiedliche Weise zu diesem Buch beigetragen und seine Fertigstellung möglich gemacht.

An erster Stelle muss John Rosenberg genannt werden. Es ist lediglich ein unglücklicher Umstand, dass John nicht einer der Autoren dieses Buches ist. Er war von Anfang an eine der treibenden Kräfte bei der Entwicklung von BlueJ, samt den dahinterstehenden Ideen und der Pädagogik, und wir haben über dieses Buchprojekt schon seit vielen Jahren geredet. Vieles von dem Material, das in diesem Buch vorgestellt wird, wurde in Diskussionen mit John entwickelt. Die schlichte Tatsache, dass ein Tag nur 24 Stunden hat, von denen zu viele schon durch zu viele andere Aufgaben belegt waren, hat ihn selbst vom Schreiben dieses Buches abgehalten. John hat fortwährend zu diesem Buch beigetragen, während es geschrieben wurde, und hat bei seiner Verbesserung an vielen Stellen geholfen. Wir haben seine Freundschaft und Zusammenarbeit sehr zu schätzen gewusst.

Einige weitere Personen haben geholfen, BlueJ seine heutige Form zu geben: Bruce Quig, Davin McCall und Andrew Patterson in Australien, Ian Utting, Poul Henriksen und Neil Brown in England. Alle arbeiten seit vielen Jahren an BlueJ, verbessern und erweitern den Entwurf und die Implementierung neben all ihren anderen Verpflichtungen. Ohne ihren Beitrag hätte BlueJ niemals seine heutige Qualität und Popularität erreicht und dieses Buch wäre möglicherweise nie geschrieben worden.

Ein weiterer wichtiger Beitrag, der BlueJ und dieses Buch erst ermöglicht hat, ist die großzügige Unterstützung zuerst durch Sun Microsystems und jetzt durch Oracle. Sun hat BlueJ jahrelang großzügig unterstützt – eine Strategie, die Oracle nach der Übernahme von Sun nicht geändert hat. Wir sind sehr dankbar für diese wichtige Hilfe.

Das Pearson-Team hat auf fantastische Weise dafür gesorgt, dass dieses Buch fertig wurde, und die schlimmste Befürchtung jedes Autors ausgeräumt – dass sein Buch unbemerkt bleiben könnte. Ein besonderer Dank geht daher an unsere Lektorin Tracy Johnson sowie an Camille Trentacoste und Carole Snyder, die uns beim Schreiben in jedweder Hinsicht unterstützt haben.

David möchte seinen persönlichen Dank an die Mitarbeiter und Studenten der School of Computing an der Universität von Kent richten. Es war immer ein großes Privileg, die Studenten der einführenden OO-Veranstaltung unterrichten zu dürfen. Sie liefern den entscheidenden Ansporn und die Motivation, durch die das Lehren so viel Spaß macht. Ohne die wertvolle Hilfe von Kollegen und studentischen Betreuern wäre es unmöglich, Lehrveranstaltungen durchzuführen.

Michael möchte Davin, Neil und Phil danken, die eine so hervorragende Arbeit bei der Entwicklung und Wartung von BlueJ, Greenfoot und unserer Website geleistet haben. Ohne ein solch großartiges Team wäre dies nicht zu realisieren gewesen.

Projekte, die in diesem Buch detailliert besprochen werden

Figuren — Kapitel 1

Einfaches Zeichnen mit grafischen Figuren; demonstriert die Erzeugung von Objekten, Methodenaufrufe und Parameter.

Haus — Kapitel 1

Ein Beispiel, in dem mit einigen Figuren eine Zeichnung erstellt wird; führt Quelltext, Java-Syntax und Übersetzung ein.

Laborkurse — Kapitel 1, Kapitel 2, Kapitel 10

Ein einfaches Beispiel mit Kursen und Studenten; demonstriert Objekte, Datenfelder und Methoden wird erneut verwendet in Kapitel 10, um Vererbung hinzuzufügen.

Ticketautomat — Kapitel 2

Eine Simulation eines Ticketautomaten für Zugtickets; führt weiter in Datenfelder, Konstruktoren, sondierende und verändernde Methoden, Parameter und einige einfache Anweisungen ein.

Buch-Aufgabe — Kapitel 2

Speicherung von Informationen über Bücher; wiederholt die Konstrukte, die im Beispiel des Ticketautomaten benutzt werden.

Zeitanzeige — Kapitel 3

Eine Implementierung einer Anzeige einer Digitaluhr; illustriert die Konzepte Abstraktion, Modularisierung und Objektinteraktion.

Mail-System — Kapitel 3

Eine einfache Simulation eines Mail-Systems; demonstriert Objekterzeugung und Objektinteraktion.

Musiksammlung — Kapitel 4, Kapitel 13

Eine Implementierung eines Verwaltungssystems für Musik-Clips; dient zur Einführung von Sammlungen und Schleifen. Umfasst die Fähigkeit, *mp3*-Dateien abzuspielen. In Kapitel 11 wird das System um eine Benutzungsoberfläche ergänzt.

Auktion — Kapitel 4

Ein Auktionssystem; mehr über Sammlungen und Schleifen, diesmal mit Iteratoren.

Tiermonitoring — Kapitel 5, 6

Ein System, um Tierpopulationen zu beobachten bzw. zu überwachen, z.B. in Nationalparks. Dieses System wird verwendet, um funktionale Verarbeitung von Sammlungen einzuführen.

Technischer-Kundendienst — Kapitel 6, 14

Eine Implementierung eines *Eliza*-ähnlichen Dialogsystems, das den technischen Kundendienst Kunden gegenüber simulieren soll; führt in die Benutzung von Bibliotheksklassen im Allgemeinen und in einige bestimmte Klassen im Besonderen ein; beschreibt das Lesen und Schreiben von Dokumentationen.

Kritzeln — Kapitel 6

Ein Programm zum Zeichnen, das zeigt, wie man die Klassen über ihre Schnittstellen kennenlernen kann.

Baelle — Kapitel 6

Eine grafische Animation springender Bälle; demonstriert die Trennung von Schnittstelle und Implementierung und einfache grafische Operationen.

Weblog-Auswertung — Kapitel 7, 14

Ein Programm zum Analysieren einer Logdatei für Zugriffe auf Webseiten; führt Arrays und `for`-Schleifen ein.

Automat — Kapitel 7

Eine Reihe von Beispielen zellulärer Automaten. Wird verwendet, um Praxis mit Array-Programmierung zu bekommen.

Brain — Kapitel 7

Eine Version von *Brian's Brain*, die wir verwenden, um zweidimensionale Arrays zu besprechen.

Zuul — Kapitel 8, 11, 14

Ein textbasiertes, interaktives Adventure-Game; sehr gut erweiterbar, hervorragend für offene Studentenprojekte. Wird hier benutzt, um guten Klassenentwurf, Kopplung und Kohäsion zu diskutieren, und wird in Kapitel 11 noch einmal als Beispiel für die Anwendung von Vererbung aufgegriffen.

Onlineshop — Kapitel 9

Die frühen Phasen der Implementierung eines Teils einer Website für einen Onlineshop, der auch Benutzerkommentare berücksichtigt; wird zur Diskussion von Test- und Fehlersuchstrategien benutzt.

Rechner **Kapitel 9**

Eine Implementierung eines Taschenrechners. Dieses Beispiel wiederholt bereits vorgestellte Konzepte und wird für die Diskussion über Testen und Fehlerbeseitigung verwendet.

Ziegelsteine **Kapitel 9**

Eine einfache Übung zur Fehlersuche; modelliert das Beladen von Paletten mit Ziegelsteinen für einfache Berechnungen.

Netzwerk **Kapitel 10, 11**

Teil einer sozialen Netzwerk-Anwendung. Dieses Projekt wird ausführlich diskutiert und erweitert, um die Grundlagen von Vererbung und Polymorphie darzustellen.

Fuechse-und-Hasen **Kapitel 12**

Eine klassische Jäger-Beute-Simulation; wiederholt Vererbungskonzepte und führt Interfaces und abstrakte Klassen ein.

Bildbetrachter **Kapitel 13**

Eine einfache Anwendung zur Bildbetrachtung und -bearbeitung. Wir konzentrieren uns vor allem auf den Aufbau einer Benutzungsoberfläche.

Musikplayer **Kapitel 13**

Als weiteres Beispiel für den Aufbau einer Benutzungsoberfläche wird dem Projekt *Musiksammlung* aus Kapitel 4 eine GUI hinzugefügt.

Adressbuch **Kapitel 14**

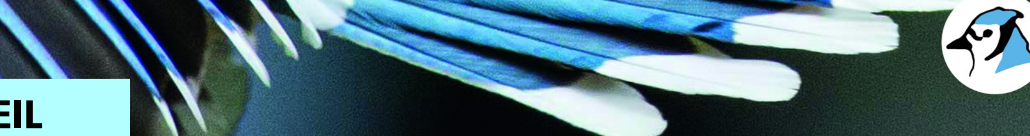
Eine Implementierung eines Adressbuches mit einer optionalen grafischen Benutzungsoberfläche. Das Nachschlagen im Adressbuch ist flexibel: Adressen können auf Basis unvollständiger Angaben des Namens oder der Telefonnummer gesucht werden. Dieses Projekt macht ausführlich Gebrauch von Exceptions.

Kinobuchungssystem **Kapitel 15**

Ein System, das Sitzplatzreservierungen für ein Kino verwaltet. Dieses Beispiel wird benutzt, um das Identifizieren von Klassen und den Entwurf einer Anwendung zu diskutieren. Es wird kein Programmtext vorgegeben, da in diesem Beispiel eine Anwendung ausgehend von einem leeren Blatt Papier entwickelt wird.

Taxi **Kapitel 16**

Das Taxi-Beispiel ist eine Kombination eines Buchungssystems, eines Verwaltungssystems und einer Simulation. Es wird als Fallstudie benutzt, um viele der Konzepte und Techniken, die im Buch vorgestellt wurden, zusammenzuführen.



TEIL

I

Objekte und Klassen

1 **Objekte und Klassen** 31

2 **Klassendefinitionen** 51

3 **Objektinteraktion** 97

4 **Objektsammlungen** 131

5 **Funktionale Verarbeitung von Sammlungen (fortgeschrittene Konzepte)** 183

6 **Bibliotheksklassen nutzen** 207

7 **Sammlungen mit fester Größe – Arrays** 263

8 **Klassenentwurf** 293

9 **Fehler vermeiden** 335

Objekte und Klassen

Lernziele

Zentrale Konzepte in diesem Kapitel: Objekte, Methoden, Klassen, Parameter

Zeit loszulegen und mit unserer Einführung in die objektorientierte Programmierung zu beginnen. Um Programmieren zu lernen, bedarf es zweierlei: ein wenig Theorie und viel, viel Praxis. Das eine wie das andere bekommen Sie in diesem Buch geboten, sodass sich beide gegenseitig unterstützen und befruchten können.

Zuerst müssen wir zwei zentrale Konzepte der Objektorientierung verstehen: *Objekte* und *Klassen*. Sie bilden die Basis jedweder objektorientierten Programmierung, gleich in welcher objektorientierten Sprache. Lassen Sie uns also mit einer kurzen Besprechung dieser beiden grundlegenden Konzepte beginnen.

1.1 Objekte und Klassen

Wenn Sie ein Computerprogramm in einer objektorientierten Sprache schreiben, dann erstellen Sie in Ihrem Computer ein Modell eines Ausschnitts der realen Welt. Dieser Ausschnitt setzt sich aus den *Objekten* zusammen, die im Anwendungsbereich vorkommen. Diese Objekte müssen im zu erstellenden Computermodell geeignet repräsentiert werden. Die Objekte des Anwendungsbereichs unterscheiden sich je nach Programm, das Sie schreiben. Es können Wörter und Absätze sein, wenn Sie eine Textverarbeitung programmieren, Nutzer und Nachrichten, wenn Sie an einem sozialen Netzwerk arbeiten, oder Monster, wenn Sie ein Computerspiel schreiben.

> **Konzept**
> Java-**Objekte** modellieren Objekte eines Anwendungsbereichs.

Objekte können klassifiziert werden. Eine Klasse beschreibt – auf abstrakte Weise – alle Objekte einer bestimmten Art.

Wir können diese etwas abstrakten Aussagen klarer machen, indem wir ein Beispiel betrachten. Nehmen Sie an, Sie möchten eine Verkehrssimulation entwerfen. Ein Begriff, mit dem Sie dann sicherlich umgehen werden, ist „Auto“. Was ist ein Auto in unserem Kontext? Ist es eine Klasse oder ein Objekt? Einige zusätzliche Fragen können uns helfen, eine Entscheidung zu treffen.

Welche Farbe hat ein Auto? Welche Höchstgeschwindigkeit hat es? Wo befindet es sich gerade?

Konzept

Objekte werden durch **Klassen** erzeugt. Eine Klasse beschreibt eine bestimmte Art von Objekten; Objekte repräsentieren individuelle Instanzen einer Klasse.

Wie Sie sicher bemerkt haben, können wir diese Fragen nicht beantworten, bevor wir über ein bestimmtes Auto sprechen. Der Grund ist, dass wir uns mit dem Wort „Auto“ auf die *Klasse* Auto beziehen – wir reden über Autos im Allgemeinen, nicht über ein bestimmtes.

Wenn ich aber sage: „Das Auto, das zu Hause in meiner Garage parkt“, dann kann ich die obigen Fragen beantworten. Das Auto ist rot, es fährt nicht sehr schnell, und es steht in meiner Garage. Jetzt rede ich über ein Objekt – über ein spezielles Exemplar eines Autos.

Wir nennen üblicherweise ein bestimmtes Objekt eine *Instanz* (*instance*, Exemplar). Diesen Begriff werden wir von nun an recht häufig verwenden. „Instanz“ ist ungefähr synonym zu „Objekt“ – wir beziehen uns auf Objekte als Instanzen, wenn wir betonen wollen, dass sie von einer bestimmten Klasse sind (wie in „dieses Objekt ist eine Instanz der Klasse `Auto`“).

Bevor wir diese etwas theoretische Diskussion fortsetzen, wollen wir zunächst ein Beispiel betrachten.

1.2 Instanzen erzeugen

Starten Sie BlueJ und öffnen Sie das Projekt *Figuren*.[1] Sie sollten daraufhin ein Fenster sehen, das dem in Abbildung 1.1 ähnelt.

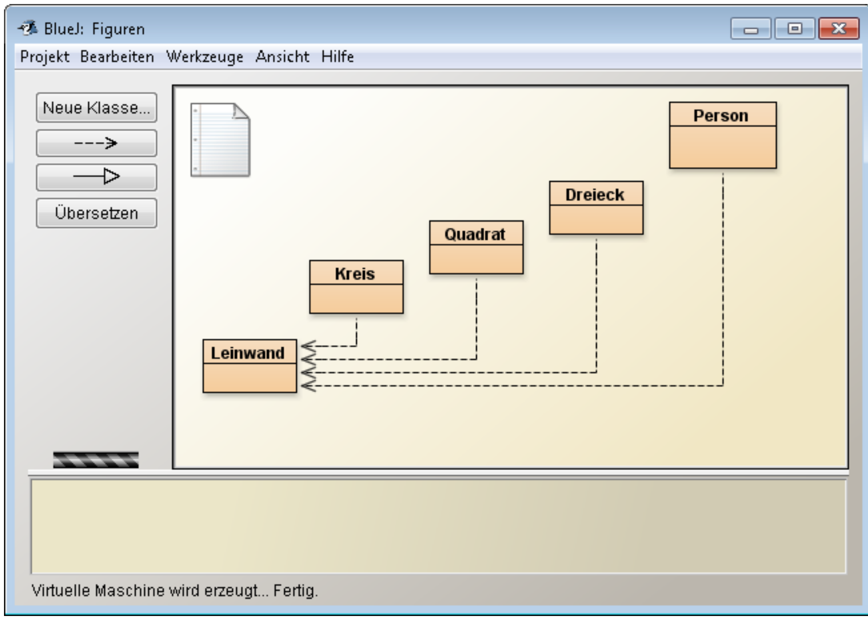


Abbildung 1.1 Das Projekt *Figuren* in BlueJ.

1 Wir empfehlen, dass Sie beim Lesen dieses Buches immer wieder selbst aktiv werden und die Übungen durcharbeiten. An dieser Stelle nehmen wir an, dass Sie bereits wissen, wie man BlueJ startet und das Beispielprojekt öffnet. Falls dies nicht der Fall ist, lesen Sie bitte zuerst Anhang A.

In diesem Fenster sollte ein Diagramm sichtbar sein. Jedes der gefärbten Rechtecke in diesem Diagramm repräsentiert eine Klasse in diesem Projekt. Die Klassen dieses Projekts heißen **Kreis**, **Quadrat**, **Dreieck**, **Person** und **Leinwand**.

Klicken Sie mit der rechten Maustaste auf die Klasse **Kreis** und wählen Sie

```
new Kreis()
```

aus dem Kontextmenü. Das System fragt Sie nun nach einem „Instanznamen" – klicken Sie auf OK, der vorgegebene Name ist vorerst gut genug. Im unteren Bereich des Fensters sollte nun ein rotes Rechteck erscheinen, das mit „kreis1" beschriftet ist (Abbildung 1.2).

Abbildung 1.2 Eine Instanz in der Objektleiste.

Gerade haben Sie Ihr erstes Objekt erzeugt! „Kreis", das rechteckige Symbol in Abbildung 1.1, repräsentiert die Klasse **Kreis**; **kreis1** ist eine frisch erzeugte Instanz dieser Klasse. Den Bereich am unteren Ende des Fensters, in dem die Instanz angezeigt wird, nennen wir die *Objektleiste*.

Konvention

Wir beginnen den Namen einer Klasse mit einem Großbuchstaben (wie in **Kreis**) und benennen Objekte mit Kleinbuchstaben (wie in **kreis1**). Dies hilft uns, die Dinge voneinander zu unterscheiden.

Übung 1.1 Erzeugen Sie einen weiteren Kreis. Erzeugen Sie dann ein Quadrat.

1.3 Methoden aufrufen

Führen Sie einen Rechtsklick auf einer der Kreis-Instanzen aus (nicht auf der Klasse!). Es erscheint ein Kontextmenü (Abbildung 1.3) mit mehreren Einträgen. Wählen Sie **sichtbarMachen** aus – dies bewirkt, dass eine Repräsentation dieses Kreises in einem eigenen Fenster angezeigt wird (Abbildung 1.4).

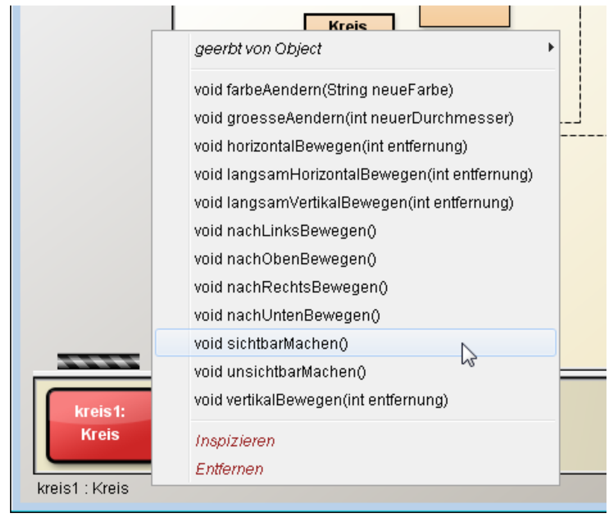


Abbildung 1.3 Das Kontextmenü eines Objekts mit der Liste seiner Operationen.

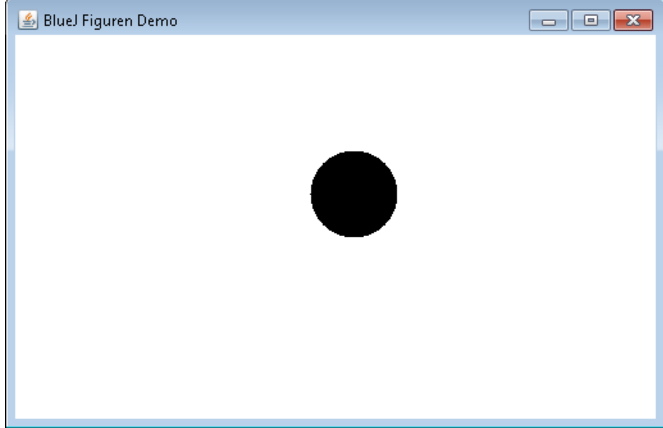


Abbildung 1.4 Die Visualisierung eines Kreises.

Konzept

Wir können mit Objekten kommunizieren, indem wir ihre **Methoden** aufrufen. Ein Objekt tut üblicherweise etwas, wenn eine seiner Methoden aufgerufen wird.

Sie sehen einige weitere Einträge im Kontextmenü des Kreises. Versuchen Sie, `nachRechtsBewegen` oder `nachUntenBewegen` auszuwählen und damit den Kreis näher an die Ränder des Fensters zu rücken. Probieren Sie auch `unsichtbarMachen` und `sichtbarMachen`, um den Kreis zu verstecken und wieder anzuzeigen.

Übung 1.2 Was geschieht, wenn Sie `nachUntenBewegen` zweimal aufrufen? Oder dreimal? Was passiert, wenn Sie `unsichtbarMachen` zweimal aufrufen?

Die Einträge im Kontextmenü der Kreis-Instanz repräsentieren Operationen, mit denen die Instanz manipuliert werden kann. Diese Operationen nennen wir Methoden. Die übliche Sprechweise ist, dass diese Methoden aufgerufen werden. Wir wollen diese Terminologie von nun an einhalten. Wir können Sie beispielsweise auffordern, „die Methode `nachRechtsBewegen` an dem Objekt `kreis1`“ aufzurufen.

1.4 Parameter

Rufen Sie nun die Methode **horizontalBewegen** auf. Es erscheint ein Dialog, der Sie um eine Eingabe bittet (Abbildung 1.5). Tippen Sie „50" ein und klicken Sie auf OK. Der Kreis wird sich daraufhin um 50 Bildschirmpunkte nach rechts bewegen.[2]

Konzept

Methoden können **Parameter** haben, mit denen zusätzliche Informationen für eine Aufgabe angegeben werden.

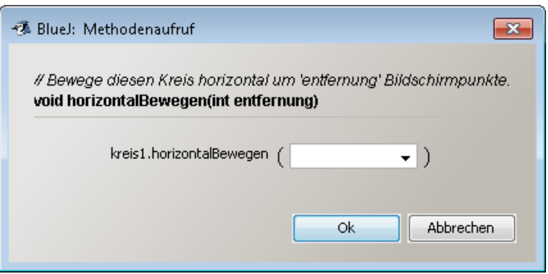


Abbildung 1.5
Ein Dialog zum Aufruf von Methoden.

Die Methode **horizontalBewegen**, die Sie gerade aufgerufen haben, ist so gestaltet, dass sie zusätzliche Informationen von Ihnen benötigt, um ihre Aufgabe erfüllen zu können. In diesem Fall ist diese zusätzliche Information, um wie viele Bildschirmpunkte der Kreis bewegt werden soll. Die Methode **horizontalBewegen** ist somit flexibler als die Methoden **nachRechtsBewegen** und **nachLinksBewegen**. Die Letzteren bewegen den Kreis immer um eine festgelegte Entfernung, wohingegen **horizontalBewegen** die Möglichkeit bietet anzugeben, wie weit der Kreis bewegt werden soll.

Übung 1.3 Bevor Sie weiterlesen, sollten Sie die Methoden **vertikalBewegen**, **langsamVertikalBewegen** und **groesseAendern** aufrufen. Finden Sie heraus, wie man **horizontalBewegen** benutzen kann, um den Kreis 70 Bildschirmpunkte nach links zu bewegen.

Die zusätzlichen Werte, die manche Methoden benötigen, werden *Parameter* genannt. Eine Methode gibt an, welche Art von Parametern sie erwartet. Wenn beispielsweise die Methode **horizontalBewegen** wie in Abbildung 1.5 aufgerufen wird, dann wird im Dialogfenster die Zeile

```
void horizontalBewegen(int entfernung)
```

im oberen Bereich angezeigt. Dies ist der *Kopf* der Methode. Der Kopf liefert Informationen über eine Methode. Der Teil zwischen den Klammern (**int entfernung**) liefert Informationen über benötigte Parameter. Für jeden Parameter wird ein Typ und ein Name definiert. Der Kopf im obigen Beispiel sagt aus, dass die Methode **horizontalBewegen** einen Parameter vom Typ **int** mit dem Namen **entfernung** erwartet. Der Name gibt dabei einen Hinweis auf die Bedeutung des Parameters. Der Name einer Methode und die Parametertypen, die im Kopf vorgefunden werden, werden zusammen als die *Signatur* der Methode bezeichnet.

Konzept

Der Name und die Parametertypen einer Methode werden als ihre **Signatur** bezeichnet. Sie benennen die benötigten Informationen für einen Aufruf der Methode.

2 Bildschirmpunkte werden auch als Pixel bezeichnet. Der Bildschirm besteht aus einem Raster von Bildschirmpunkten.

1.5 Datentypen

Konzept

Parameter haben **Typen**. Ein Typ definiert, welche Arten von Werten ein Parameter annehmen kann.

Ein Typ gibt an, welche Art von Information als Parameter übergeben werden kann. Der Typ `int` beispielsweise bezeichnet ganze Zahlen (aus dem Englischen *integer*, daher die Abkürzung „int“).

Im obigen Beispiel besagt die Signatur von `horizontalBewegen`, dass wir, bevor die Methode ausgeführt werden kann, eine ganze Zahl festlegen müssen, die die Entfernung angibt. Das Eingabefeld in Abbildung 1.5 ermöglicht diese Eingabe.

In den bisherigen Beispielen haben wir nur den Datentyp `int` kennengelernt. Die Parameter der Methoden zum Bewegen und der Methode `groesseAendern` sind alle von diesem Typ.

Eine nähere Betrachtung des Kontextmenüs einer Instanz zeigt, dass die Einträge in diesem Menü bereits die Parameterinformation enthalten. Wenn eine Methode keine Parameter hat, folgen auf den Namen nur leere Klammern. Wenn sie einen Parameter hat, wird der Typ dieses Parameters angezeigt. In der Liste der Methoden für einen Kreis können Sie eine Methode mit einem anderen Parametertyp entdecken: Die Methode `farbeAendern` hat einen Parameter vom Typ `String`.

Der Typ `String` signalisiert, dass ein Stück Text (beispielsweise ein Wort oder ein Satz) erwartet wird. Da ein solcher Text sich aus einzelnen Zeichen zusammensetzt, bezeichnen wir ihn als eine Zeichenkette. Solche Zeichenketten werden in doppelten Anführungsstrichen angegeben. Um beispielsweise das Wort *rot* als Zeichenkette zu übergeben, tippen Sie:

```
"rot"
```

Der Aufrufdialog enthält außerdem über der Zeile mit dem Methodenkopf einen Textabschnitt, den wir als Kommentar bezeichnen. *Kommentare* werden angegeben, um dem menschlichen Leser Informationen zu geben; sie werden ausführlicher in Kapitel 2 behandelt. Der Kommentar der Methode `farbeAendern` beschreibt beispielsweise, welche Farbnamen dem System bekannt sind.

Übung 1.4 Rufen Sie die Methode `farbeAendern` an einer Ihrer Kreis-Instanzen auf und übergeben Sie die Zeichenkette `"rot"`. Dies sollte die Farbe des Kreises ändern. Versuchen Sie es auch mit anderen Farben.

Übung 1.5 Dies ist ein sehr einfaches Beispiel, es werden nicht sehr viele Farben unterstützt. Was passiert, wenn Sie eine Farbe angeben, die nicht unterstützt wird?

Übung 1.6 Rufen Sie die Methode `farbeAendern` auf und geben Sie die Farbe ohne die Anführungszeichen an. Was passiert?

Fallstrick

Ein typischer Fehler von Programmieranfängern ist, dass sie bei Angabe eines Parameters vom Typ `String` die doppelten Anführungszeichen vergessen. Wenn Sie beispielsweise `blau` statt `"blau"` tippen, bekommen Sie eine Fehlermeldung in der Art: "Error: cannot resolve symbol – variable blau".

Java unterstützt etliche weitere Datentypen, einschließlich Typen für Fließkommazahlen und einzelne Zeichen. Wir werden diese hier nicht alle diskutieren, sondern zu einem späteren Zeitpunkt auf sie zurückkommen. Wenn Sie jetzt mehr darüber wissen wollen, dann können Sie sich Anhang B ansehen.

1.6 Eine Klasse, viele Instanzen

Sobald Sie eine Klasse haben, können Sie beliebig viele Instanzen dieser Klasse erzeugen. Mit der Klasse `Kreis` können Sie beliebig viele Kreise erzeugen, mit der Klasse `Quadrat` beliebig viele Quadrate.

Übung 1.7 Erzeugen Sie mehrere Kreis-Instanzen auf der Objektleiste, indem Sie mehrfach den Eintrag `new Kreis()` aus dem Kontextmenü der Klasse `Kreis` auswählen. Machen Sie die Instanzen sichtbar und bewegen Sie sie mit den Bewegungsmethoden. Machen Sie einen Kreis groß und gelb, einen anderen klein und grün. Probieren Sie auch die anderen Figuren: Erzeugen Sie ein paar Dreiecke und Quadrate. Ändern Sie deren Positionen, Größen und Farben.

Jedes dieser Objekte hat seine eigene Position, Farbe und Größe. Sie ändern eine Eigenschaft eines Objekts (etwa seine Farbe), indem Sie eine Methode an dem Objekt aufrufen. Dieser Aufruf beeinflusst dieses spezielle Objekt, aber keine anderen Objekte.

Konzept

Eine Klasse, **viele Instanzen:** Von einer Klasse können viele gleichartige Instanzen erzeugt werden.

Sie haben möglicherweise auch ein weiteres Detail über Parameter entdeckt. Sehen Sie sich die Methode `groesseAendern` an einem Dreieck an. Der Methodenkopf lautet:

```
void groesseAendern (int neueHoehe, int neueBreite)
```

Dies ist ein Beispiel für eine Methode mit mehr als einem Parameter. Diese Methode hat zwei Parameter, die im Kopf durch ein Komma voneinander getrennt sind. Allgemein kann eine Methode beliebig viele Parameter haben.

1.7 Zustand

Konzept

Objekte haben einen **Zustand**. Dieser Zustand wird durch Werte repräsentiert, die in Datenfeldern gehalten werden.

Die Menge der momentanen Werte der Attribute, die ein Objekt definieren (wie *x*- und *y*-Position, Farbe, Durchmesser und Sichtbarkeit eines Kreises), wird auch als der *Zustand* des Objekts bezeichnet. Dies ist ein weiteres Beispiel der allgemein üblichen Terminologie, an die wir uns von nun an halten werden.

In BlueJ kann der Zustand eines Objekts genauer untersucht werden, indem der Eintrag INSPIZIEREN aus dem Kontextmenü einer Instanz ausgewählt wird. Wenn ein Objekt auf diese Weise untersucht wird, erscheint ein Fenster, das dem in Abbildung 1.6 ähnlich ist. Dieses Fenster nennen wir den *Objektinspektor*.

Abbildung 1.6 Ein Objektinspektor, der Details eines Objekts anzeigt.

Übung 1.8 Sorgen Sie dafür, dass mehrere Objekte auf der Objektleiste liegen, und untersuchen Sie diese der Reihe nach. Versuchen Sie, den Zustand eines Objekts zu ändern (beispielsweise durch den Aufruf von `nachLinksBewegen`), während der Objektinspektor geöffnet ist. Sie sollten beobachten können, wie sich die Werte im Objektinspektor ändern.

Einige Methoden verändern den Zustand eines Objekts, wenn sie aufgerufen werden. Die Methode `nachLinksBewegen` ändert beispielsweise das Attribut `xPosition`. In Java werden diese Attribute von Objekten häufig *Datenfelder* oder einfach nur Felder (*fields*) genannt.

1.8 Das Innenleben eines Objekts

Beim Untersuchen verschiedener Objekte stellen wir fest, dass alle Instanzen *derselben* Klasse auch die gleichen Datenfelder haben. Das heißt, Anzahl, Typen und Namen der Datenfelder sind gleich, während der aktuelle Wert eines Datenfelds in jedem Objekt anders sein kann. Im Unterschied dazu haben Instanzen *unterschiedlicher* Klassen auch unterschiedliche Datenfelder. Ein Kreis beispielsweise hat ein Datenfeld „durchmesser", während ein Dreieck die Datenfelder „breite" und „hoehe" hat.

Dies ist deshalb so, weil Anzahl, Typen und Namen von Datenfeldern durch die Klasse definiert sind, nicht in einem Objekt. Die Klasse `Kreis` definiert, dass jede

Kreis-Instanz fünf Datenfelder hat, und zwar **durchmesser**, **xPosition**, **yPosition**, **farbe** und **istSichtbar**. Sie legt auch fest, welche Typen diese Datenfelder haben. Die ersten drei sind vom Typ **int**, die Farbe ist vom Typ **String** und **istSichtbar** ist vom Typ **boolean**. (Eine boolesche Variable kann einen von zwei Werten annehmen: **true** und **false**. Der Typ **boolean** wird später noch ausführlicher diskutiert.)

Wenn eine Instanz der Klasse **Kreis** erzeugt wird, dann hat diese Instanz automatisch diese Datenfelder. Die Werte der Datenfelder werden in der Instanz gespeichert. Dies garantiert, dass jede Instanz beispielsweise eine Farbe hat, ermöglicht aber auch, dass jede Instanz eine andere Farbe haben kann (Abbildung 1.7).

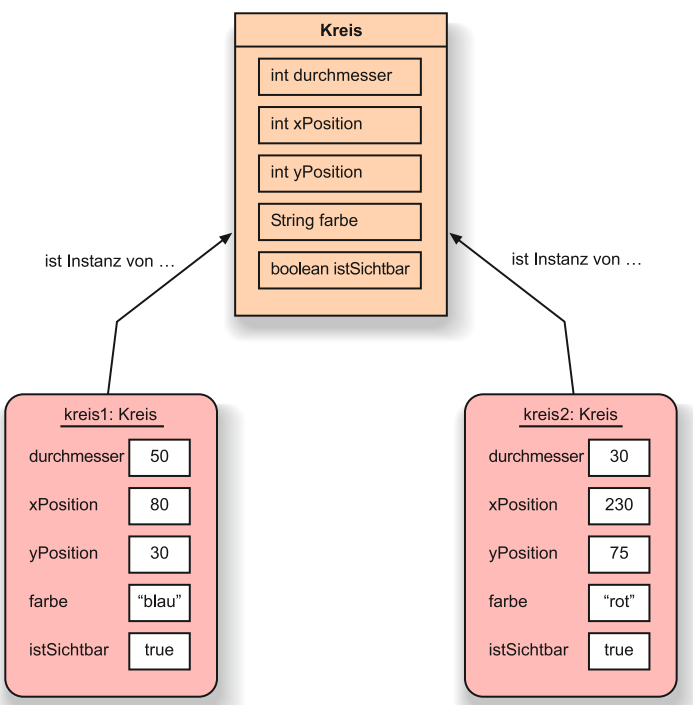


Abbildung 1.7
Eine Klasse und ihre Instanzen mit Datenfeldern und Werten.

Das Gleiche gilt für Methoden. Die Methoden werden in der Klasse eines Objekts definiert. Folglich haben alle Instanzen einer gegebenen Klasse die gleichen Methoden. Aber die Methoden werden an konkreten Instanzen aufgerufen. Auf diese Weise ist eindeutig, welches Objekt seinen Zustand ändern muss, wenn beispielsweise die Methode **nachRechtsBewegen** aufgerufen wird.

Übung 1.9 In Abbildung 1.8 auf Seite 42 sehen Sie zwei Bilder. Suchen Sie sich eines dieser Bilder aus und versuchen Sie, es mithilfe der Figuren aus dem Projekt *Figuren* nachzustellen. Notieren Sie gleichzeitig, was Sie im Einzelnen tun müssen, um dies zu erreichen. Könnten Sie es auch auf andere Weise erreichen?

1.9 Java-Code

Wenn wir in Java programmieren, schreiben wir eigentlich Befehle nieder, um Methoden an Objekten aufzurufen – so wie wir es oben für unsere Figur-Objekte gemacht haben. Allerdings tun wir dies nicht interaktiv, indem wir mit der Maus Methoden aus einem Menü auswählen, sondern indem wir die Befehle als Text eintippen. In der BlueJ-Konsole können wir uns ansehen, wie diese Befehle als Text aussehen.

Übung 1.10 Wählen Sie im Menü ANSICHT den Befehl KONSOLE ANZEIGEN. Damit wird ein anderes Fenster aufgerufen, das BlueJ zur Ausgabe von Text verwendet. Wählen Sie in diesem Terminalfenster den Befehl METHODENAUFRUFE PROTOKOLLIEREN im Menü OPTIONEN. Mit dieser Funktion werden unsere Methodenaufrufe (als Text) in der Konsole angezeigt. Erzeugen Sie jetzt einige Objekte, rufen Sie einige ihrer Methoden auf und beobachten Sie die Ausgabe im Konsolenfenster.

Mithilfe des Befehls METHODENAUFRUFE PROTOKOLLIEREN können wir erkennen, dass die Folge, ein Personenobjekt zu erzeugen und seine Methoden **sichtbarMachen** und **nachRechtsBewegen** aufzurufen, als Java-Code in Textform wie folgt aussieht:

```
Person person1 = new Person();
person1.sichtbarMachen();
person1.nachRechtsBewegen();
```

An diesem Text lassen sich einige Dinge beobachten:

- Wir sehen, wie der Code aussieht, wenn ein Objekt erzeugt und mit einen Namen verbunden wird. Technisch gesehen *speichern wir das Person-Objekt in einer Variablen*. Was dies genau bedeutet, besprechen wir ausführlich im nächsten Kapitel.
- Wir sehen, dass wir für den Aufruf einer Methode an einem Objekt den Namen des Objekts gefolgt von einem Punkt und dem Namen der Methode schreiben. Der Befehl endet mit einer Parameterliste beziehungsweise einem leeren Paar Klammern, wenn es keine Parameter gibt.
- Alle Java-Anweisungen enden mit einem Semikolon.

Doch wir wollen nicht nur Java-Anweisungen betrachten, sondern auch welche eingeben. Hierzu verwenden wir die *Direkteingabe*. (Sie können jetzt die Funktion METHODENAUFRUFE PROTOKOLLIEREN ausschalten und die Konsole schließen.)

Übung 1.11 Wählen Sie im Menü ANSICHT den Befehl DIREKTEINGABE ANZEIGEN aus. Es sollte ein neues Fenster neben der Objektleiste in Ihrem BlueJ-Hauptfenster erscheinen, bei dem es sich um die *Direkteingabe* handelt. Hier können Sie Ihren Java-Code eingeben.

In der Direkteingabe können Sie Java-Code eingeben, der das Gleiche macht wie der Code, den wir zuvor interaktiv erstellt haben.

Übung 1.12 Geben Sie in der Direkteingabe den obigen Code ein, um ein `Person`-Objekt zu erzeugen und ihre Methoden `sichtbarMachen` und `nachRechtsBewegen` aufzurufen.

Das Eintippen dieser Befehle sollte die gleiche Wirkung haben wie der Aufruf des gleichen Befehls aus dem Menü des Objekts. Wenn Sie stattdessen eine Fehlermeldung erhalten, dann haben Sie beim Befehl einen Tippfehler gemacht. Achten Sie also auf Ihre Schreibweise. Sie werden feststellen, dass auch nur ein einziges falsches Zeichen reicht, damit der Befehl nicht funktioniert

Tipp

Sie können in der Direkteingabe unter Verwendung des Aufwärts-Pfeils bereits verwendete Befehle erneut aufrufen.

1.10 Objektinteraktion

Für den nächsten Abschnitt werden wir mit einem anderen Projekt arbeiten. Schließen Sie das Projekt *Figuren*, falls Sie es noch geöffnet haben, und öffnen Sie das Projekt *Haus*.

Übung 1.13 Öffnen Sie das Projekt *Haus*. Erzeugen Sie eine Instanz der Klasse `Bild` und rufen Sie an ihr die Methode `zeichne` auf. Probieren Sie dann die Methoden `inSchwarzWeissAendern` und `inFarbeAendern` aus.

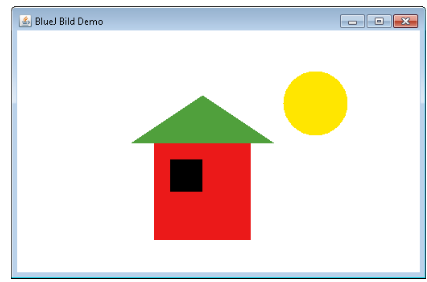
Übung 1.14 Wie erzeugt Ihrer Meinung nach die Klasse `Bild` die Zeichnung?

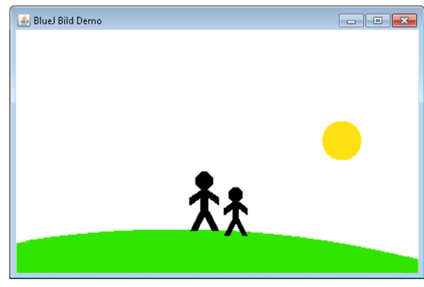
Fünf der Klassen in diesem Projekt sind identisch mit den Klassen aus dem Projekt *Figuren*. Hier haben wir eine zusätzliche Klasse: `Bild`. Diese Klasse ist so programmiert, dass sie automatisch das tut, was Sie manuell in ***Übung 1.9*** getan haben.

In der Praxis würden wir, wenn wir eine Folge von Aufgaben erledigt haben möchten, dies nicht von Hand tun. Stattdessen würden wir eine Klasse erzeugen, die die Arbeit für uns erledigt. Das genau macht die Klasse `Bild`.

Die Klasse `Bild` ist so geschrieben, dass durch den Aufruf der Methode `zeichne` zwei Quadrat-Instanzen (eines für die Wand, eines für das Fenster) sowie ein Dreieck und ein Kreis erzeugt werden. Wenn anschließend die Methode `zeichne` aufgerufen wird, dann werden diese Instanzen so lange bewegt und in Farbe und Form verändert, bis die Figuren so aussehen, wie in Abbildung 1.8 dargestellt.

Abbildung 1.8
Zwei Zeichnungen, die mit einigen Figuren erstellt wurden.





Konzept

Methodenaufrufe: Objekte können miteinander kommunizieren, indem sie gegenseitig ihre **Methoden aufrufen**.

Die entscheidenden Punkte an dieser Stelle sind: Objekte können andere Objekte erzeugen und sie können die Methoden dieser Objekte aufrufen. In einem normalen Java-Programm können es sehr schnell Hunderte oder Tausende von Objekten werden. Der Benutzer eines Programms startet es lediglich (indem üblicherweise ein erstes Objekt erzeugt wird), und alle weiteren Objekte werden direkt oder indirekt durch dieses Objekt erzeugt.

Die große Frage ist nun: Wie schreiben wir die Klasse für solch ein Objekt?

1.11 Quelltext

Konzept

Der **Quelltext** einer Klasse legt die Struktur und das Verhalten (die Datenfelder und die Methoden) aller Instanzen dieser Klasse fest.

Jeder Klasse ist ein *Quelltext* zugeordnet. Der Quelltext ist ein Text, der die Details einer Klasse beschreibt. In BlueJ kann der Quelltext einer Klasse betrachtet werden, indem aus dem Kontextmenü der Klasse der Eintrag BEARBEITEN aufgerufen oder ein Doppelklick auf dem Klassensymbol ausgeführt wird.

Übung 1.15 Aktivieren Sie das Kontextmenü der Klasse `Bild`. Sie sollten einen Eintrag BEARBEITEN sehen – wählen Sie diesen aus. Dies startet einen Texteditor, der den Quelltext der Klasse anzeigt.

Der Quelltext ist in der Programmiersprache Java formuliert. Er definiert, welche Datenfelder und Methoden eine Klasse hat, und legt exakt fest, was passiert, wenn eine Methode aufgerufen wird. Im nächsten Kapitel werden wir ausführlich darauf eingehen, was der Quelltext einer Klasse umfasst und wie er strukturiert ist.

Um die Kunst des Programmierens zu erlernen, muss man in erster Linie lernen, solche Definitionen von Klassen zu erstellen. Wir verwenden zu diesem Zweck die Programmiersprache Java (obwohl es viele andere Programmiersprachen gibt, in denen Programme geschrieben werden können).

Wenn Sie eine Änderung am Quelltext vornehmen und den Editor schließen,[3] erscheint das Symbol für die Klasse im Diagramm gestreift. Die Streifen zeigen an, dass der Quelltext geändert wurde. Die Klasse muss nun übersetzt werden, indem der ÜBERSETZEN-Knopf gedrückt wird (falls Sie mehr darüber erfahren möchten, was

3 In BlueJ ist es nicht notwendig, vor dem Schließen des Editors den Text explizit zu speichern. Beim Schließen wird der Quelltext automatisch gespeichert.

beim Übersetzen passiert, dann lesen Sie den Abschnitt **Programmübersetzung**). Sobald die Klasse übersetzt ist, können Instanzen von ihr erzeugt werden, um an ihnen die Änderungen auszuprobieren.

Programmübersetzung

Wenn Menschen Computerprogramme schreiben, benutzen sie typischerweise eine „höhere" Programmiersprache wie beispielsweise Java. Ein Problem dabei ist, dass ein Computer den Quelltext von Java nicht direkt ausführen kann. Java wurde entworfen, um einigermaßen lesbar für Menschen zu sein, nicht für Computer. Computer arbeiten intern mit einer binären Repräsentation eines Maschinencodes, der sich von Java erheblich unterscheidet. Das Problem ist nun, dass für uns der Maschinencode so unleserlich ist, dass wir ihn nicht direkt schreiben wollen. Wir schreiben lieber in Java. Was ist also zu tun?

Die Lösung ist ein spezielles Programm, der Compiler. Ein Compiler übersetzt Java-Text in Maschinencode. Wir können Java schreiben, den Compiler starten – dieser generiert den Maschinencode – und der Computer kann den Maschinencode ausführen. Als Konsequenz ergibt sich, dass wir jedes Mal, wenn wir den Quelltext ändern, zuerst den Compiler laufen lassen müssen, bevor wir die Klasse zum Erzeugen einer Instanz benutzen können. Andernfalls existiert nicht die Version des Maschinencodes, die der Computer braucht.

Übung 1.16 Finden Sie im Quelltext der Klasse `Bild` den Abschnitt, der das eigentliche Zeichnen ausführt. Ändern Sie ihn so, dass die Sonne blau statt gelb ist.

Übung 1.17 Fügen Sie der Zeichnung eine zweite Sonne hinzu. Dazu sollten Sie sich die Definitionen der Datenfelder zu Beginn der Klassendefinition ansehen. Sie sehen folgendermaßen aus:

```
private Quadrat wand;
private Quadrat fenster;
private Dreieck dach;
private Kreis sonne;
```

An dieser Stelle kann eine Zeile für die zweite Sonne eingefügt werden. Beispielsweise:

```
private Kreis sonne2;
```

Schreiben Sie dann auch die entsprechenden Anweisungen für die zweite Sonne an zwei unterschiedlichen Stellen und machen Sie diese sichtbar, wenn das Bild gemalt wird.

Übung 1.18 *Zusatzaufgabe*. (Dies heißt, dass diese Übung eventuell nicht schnell zu lösen ist. Wir erwarten nicht, dass jeder diese Aufgabe zu diesem Zeitpunkt lösen kann. Wenn Sie es schaffen – großartig. Wenn nicht, dann ist das kein Problem. Die Dinge werden klarer werden, sobald Sie weitergelesen haben. Kommen Sie später auf diese Übung zurück.) Fügen Sie der Version von `Bild` mit nur einer Sonne einen Sonnenuntergang hinzu. Also: Lassen Sie die Sonne langsam untergehen. *Hinweis*: Die Klasse `Kreis` hat eine Methode `langsamVertikalBewegen`, die Sie zu diesem Zweck verwenden können.

Übung 1.19 *Zusatzaufgabe*. Wenn Sie den Sonnenuntergang am Ende der Methode `zeichne` eingefügt haben (sodass die Sonne automatisch untergeht, sobald das Bild gezeichnet ist), so ändern Sie dies nun. Für den Sonnenuntergang soll eine eigene Methode `sonnenuntergang` definiert werden. Auf diese Weise können wir zuerst die Methode `zeichne` aufrufen, um die Zeichnung mit der Sonne zu sehen, und anschließend `sonnenuntergang`, um die Sonne untergehen zu lassen.

Übung 1.20 *Zusatzaufgabe*. Erzeugen Sie eine Person, die nach dem Sonnenuntergang auf das Haus zugeht.

1.12 Ein weiteres Beispiel

In diesem Kapitel haben wir bereits etliche neue Konzepte diskutiert. Um das Verständnis dieser Konzepte zu vertiefen, wollen wir sie noch einmal aus einem anderen Zusammenhang heraus betrachten. Dazu verwenden wir ein anderes Beispiel. Schließen Sie das Projekt *Haus*, falls Sie es noch geöffnet haben, und öffnen Sie das Projekt *Laborkurse*.

Dieses Projekt ist ein vereinfachter Teil einer Studentendatenbank, die entworfen wurde, um Studenten in Laborkursen zu verwalten und Kurslisten auszudrucken.

Übung 1.21 Erzeugen Sie eine Instanz der Klasse `Student`. Sie werden feststellen, dass Sie nicht nur nach dem Namen für die Instanz gefragt werden, sondern auch nach weiteren Parametern. Füllen Sie die Felder aus und klicken Sie auf OK. (Zur Erinnerung: Parameter vom Typ `String` müssen mit doppelten Anführungsstrichen angegeben werden.)

1.13 Aufrufergebnisse

Wie schon zuvor können Sie mehrere Instanzen erzeugen. Und wie zuvor haben die Instanzen Methoden, die jeweils aus ihrem Kontextmenü heraus aufgerufen werden können.

Übung 1.22 Erzeugen Sie einige Instanzen der Klasse `Student`. Rufen Sie die Methode `gibName` an jeder Instanz auf. Erklären Sie, was passiert.

Wenn wir die Methode `gibName` aufrufen, bemerken wir etwas Neues: Methoden können ein Ergebnis liefern. Tatsächlich sagt uns der Kopf einer Methode, ob diese ein Ergebnis liefert oder nicht und von welchem Typ das Ergebnis ist. Der Kopf von `gibName` (dargestellt im Kontextmenü einer Instanz) ist definiert als

```
String gibName()
```

Das Wort `String` vor dem Namen der Methode spezifiziert den Ergebnistyp. In diesem Fall wird ausgesagt, dass der Aufruf dieser Methode ein Ergebnis vom Typ `String` liefert. Die Signatur von `nameAendern` lautet:

```
void nameAendern(String neuerName)
```

Das Schlüsselwort `void` sagt an dieser Stelle, dass diese Methode kein Ergebnis liefert.

Methoden mit einem Ergebnistyp ermöglichen es uns, durch ihren Aufruf Informationen von einem Objekt abzufragen. Dies bedeutet, wir können Methoden entweder benutzen, um den Zustand eines Objekts zu ändern oder um etwas über seinen Zustand herauszufinden. Der Ergebnistyp einer Methode ist nicht Teil ihrer Signatur.

Konzept

Ergebnis: Methoden können Informationen über ein Objekt durch einen **Ergebniswert** zurückliefern.

1.14 Objekte als Parameter

Übung 1.23 Erzeugen Sie eine Instanz der Klasse `Laborkurs`. Die Signatur besagt dabei, dass Sie eine Maximalzahl für die Studenten in dem Kurs angeben müssen (eine ganze Zahl).

Übung 1.24 Rufen Sie die Methode `anzahlStudenten` an dem Laborkurs auf. Was tut sie?

Übung 1.25 Betrachten Sie die Signatur der Methode `trageStudentEin`. Sie werden bemerken, dass der Typ des erwarteten Parameters `Student` ist. Sorgen Sie dafür, dass zwei oder drei Instanzen der Klasse `Student` und eine Instanz von `Laborkurs` auf der Objektleiste liegen. Rufen Sie dann die Methode `trageStudentEin` an der `Laborkurs`-Instanz auf. Mit dem Eingabefokus im Eingabefeld klicken Sie nun auf eine der Instanzen von `Student` – dies trägt den Namen der Instanz in das Eingabefeld ein (Abbildung 1.9). Klicken Sie auf OK und der Student ist in den Laborkurs eingetragen. Tragen Sie weitere Studenten auf diese Weise ein.

Übung 1.26 Rufen Sie die Methode `listeAusgeben` an dem `Laborkurs`-Objekt auf. Es erscheint eine Liste aller Kursteilnehmer auf der BlueJ-Konsole (Abbildung 1.10).

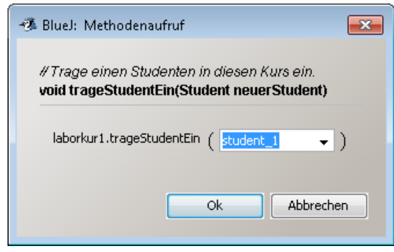


Abbildung 1.9 Einen Studenten in einen Laborkurs eintragen.

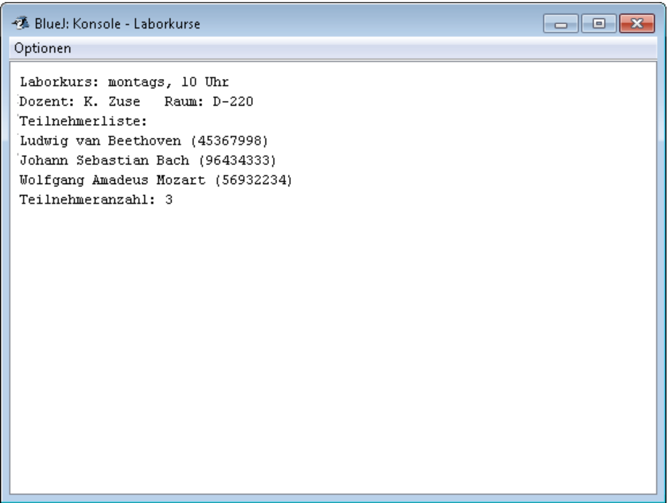


Abbildung 1.10 Ausgabe der Teilnehmerliste einer `Laborkurs`-Instanz.

Diese Übung zeigt, dass Objekte als Parameter an andere Objekte übergeben werden können. Wenn eine Methode ein Objekt als Parameter erwartet, dann wird der Name der Klasse des Objekts als Parametertyp in der Signatur angegeben.

Sehen Sie sich dieses Projekt etwas genauer an. Versuchen Sie, die Konzepte aus dem Projekt *Figuren* in diesem Zusammenhang wiederzuerkennen.

Übung 1.27 Erzeugen Sie drei Studenten mit den folgenden Eigenschaften:

Schneewittchen, Matrikelnummer: 100234, Scheine: 4

Lisa Simpson, Matrikelnummer: 122044, Scheine: 9

Charlie Brown, Matrikelnummer 12003P, Scheine: 1

Tragen Sie die drei anschließend in einen Laborkurs ein und lassen Sie die Kursliste ausgeben.

Übung 1.28 Verwenden Sie den Objektinspektor an dem `Laborkurs`, um herauszufinden, welche Datenfelder er hat.

Übung 1.29 Setzen Sie für einen Laborkurs den Dozenten, den Raum und die Anfangszeit fest. Überprüfen Sie anschließend in der Kursliste in der Konsole, ob diese Informationen angezeigt werden.

Zusammenfassung

In diesem Kapitel haben wir die Grundlagen von Klassen und Objekten untersucht. Wir haben gesehen, dass Objekte durch Klassen definiert werden. Klassen repräsentieren das generelle Konzept eines Gegenstands, während Objekte konkrete Instanzen einer Klasse repräsentieren. Wir können von jeder Klasse beliebig viele Instanzen erzeugen.

Objekte haben Methoden, damit wir mit ihnen kommunizieren können. Wir können eine Methode benutzen, um den Zustand eines Objekts zu ändern oder um Informationen über ein Objekt zu bekommen. Methoden können Parameter haben, und Parameter haben Typen. Methoden definieren einen Ergebnistyp, der festlegt, was sie zurückliefern. Wenn der Ergebnistyp `void` ist, dann liefern sie nichts zurück.

Objekte speichern Daten in ihren Datenfeldern (die auch Typen haben). Die Werte, die in den Datenfeldern eines Objekts gehalten werden, definieren seinen Zustand.

Objekte werden auf der Basis von Klassendefinitionen erzeugt, die in einer bestimmten Programmiersprache formuliert sind. Der überwiegende Anteil der Programmierung in Java beschäftigt sich mit dem Erstellen von Klassendefinitionen. Ein großes Java-Programm besteht aus vielen Klassen, jede mit vielen Methoden, die sich gegenseitig auf vielfältige Weise aufrufen.

Um Java-Programme schreiben zu können, müssen wir lernen, wie man Klassendefinitionen mit Datenfeldern und Methoden schreibt und wie man diese Klassen geeignet miteinander kombiniert. Der Rest dieses Buches wird sich genau damit beschäftigen.

NEUE BEGRIFFE IN DIESEM KAPITEL

Objekt, Klasse, Instanz, Methode, Signatur, Parameter, Typ, Zustand, Quelltext, Ergebniswert, Compiler

Zusammenfassung der Konzepte

- **Objekt** Java-Objekte modellieren Objekte eines Anwendungsbereichs.
- **Klasse** Objekte werden durch Klassen erzeugt. Eine Klasse beschreibt eine bestimmte Art von Objekten; Objekte repräsentieren individuelle Instanzen einer Klasse.
- **Methode** Wir können mit Objekten kommunizieren, indem wir ihre Methoden aufrufen. Ein Objekt tut üblicherweise etwas, wenn eine seiner Methoden aufgerufen wird.
- **Parameter** Methoden können Parameter haben, mit denen zusätzliche Informationen für eine Aufgabe angegeben werden.
- **Signatur** Der Name und die Parametertypen einer Methode werden als deren Signatur bezeichnet. Sie benennt die benötigten Informationen für einen Aufruf der Methode.
- **Typ** Parameter haben Typen. Ein Typ definiert, welche Arten von Werten ein Parameter annehmen kann.
- **Eine Klasse, viele Instanzen** Von einer Klasse können viele gleichartige Instanzen erzeugt werden.
- **Zustand** Objekte haben einen Zustand. Dieser Zustand wird durch Werte repräsentiert, die in Datenfeldern gehalten werden.
- **Methodenaufrufe** Objekte können miteinander kommunizieren, indem sie gegenseitig ihre Methoden aufrufen.
- **Quelltext** Der Quelltext einer Klasse legt die Struktur und das Verhalten (die Datenfelder und die Methoden) aller Instanzen dieser Klasse fest.
- **Ergebnis** Methoden können Informationen über ein Objekt durch einen Ergebniswert zurückliefern.

Übung 1.30 In diesem Kapitel haben wir die Datentypen `int` und `String` erwähnt. Java hat einige weitere vordefinierte Datentypen. Finden Sie heraus, welche das sind und welchem Zweck sie dienen. Sie können dies in Anhang B nachlesen oder in einem anderen Buch über Java oder in einem Onlinehandbuch zu Java. Ein englisches Onlinehandbuch ist beispielsweise zu finden unter:

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html

Übung 1.31 Welche Typen haben die folgenden Werte?

```
0
"hallo"
101
-1
true
"33"
3.1415
```

Übung 1.32 Was müssten Sie tun, um einem Kreis-Objekt ein neues Datenfeld, beispielsweise `name`, hinzuzufügen?

Übung 1.33 Schreiben Sie die Signatur für eine Methode `senden`, die einen Parameter vom Typ `String` bekommt und keinen Wert zurückliefert.

Übung 1.34 Schreiben Sie den Kopf für eine Methode `mittelwert`, die zwei Parameter vom Typ `int` bekommt und einen `int`-Wert zurückliefert.

Übung 1.35 Sehen Sie sich das Buch an, das Sie gerade lesen. Ist es ein Objekt oder eine Klasse? Wenn es eine Klasse ist, dann nennen Sie einige Objekte. Wenn es ein Objekt ist, dann nennen Sie seine Klasse.

Übung 1.36 Kann ein Objekt verschiedene Klassen haben? Diskutieren Sie.

KAPITEL 2 Klassendefinitionen

Lernziele

Zentrale Konzepte in diesem Kapitel: Datenfelder, sondierende und verändernde Methoden, Konstruktoren, Zuweisung und bedingte Anweisung, Parameter

Java-Konstrukte in diesem Kapitel: Datenfeld, Konstruktor, Kommentar, Parameter, Zuweisung (`=`), Block, `return`, `void`, zusammengesetzter Zuweisungsoperator (`+=`, `-=`), `if`

In diesem Kapitel befassen wir uns erstmals gründlich mit dem Quelltext einer Klasse. Wir werden die Basiselemente von Klassendefinitionen diskutieren: *Datenfelder*, *Konstruktoren* und *Methoden*. Methoden enthalten Anweisungen. Wir werden zuerst Methoden betrachten, die lediglich einfache arithmetische Anweisungen und Ausgabeanweisungen enthalten. Später führen wir *bedingte Anweisungen* ein, mit denen eine Auswahl zwischen verschiedenen Aktionen innerhalb einer Methode getroffen werden kann.

Wir beginnen, indem wir uns detailliert ein neues Projekt ansehen. Dieses Projekt enthält eine naive Implementierung eines Ticketautomaten. Während wir die elementaren Konzepte von Klassen einführen, werden wir schnell merken, dass diese Implementierung einige Schwächen hat. Entsprechend werden wir anschließend eine weiterentwickelte Version des Ticketautomaten beschreiben, die etliche Verbesserungen aufweist. Schließlich, um die Konzepte dieses Kapitels noch einmal zu wiederholen, werfen wir einen Blick auf die interne Struktur der *Laborkurse* aus dem letzten Kapitel.

2.1 Ticketautomaten

Auf Bahnhöfen und U-Bahnstationen stehen üblicherweise Ticketautomaten, die ein Ticket ausdrucken, sobald ein Kunde den korrekten Betrag eingeworfen hat. In diesem Kapitel werden wir eine Klasse definieren, die einen solchen Automaten modelliert. Da wir hier erstmalig die Struktur einer Klasse untersuchen, ist die Simulation selbst zu Anfang absichtlich simpel gehalten. Auf diese Weise können wir untersuchen, inwieweit solche Modelle von ihren Originalen in der Realität abwei-

chen und wie wir unsere Klassen ändern müssen, damit sich die erzeugten Objekte ihren realen Vorlagen annähern.

Unser Ticketautomat arbeitet folgendermaßen: Ein Kunde wirft Geld ein und fordert ein Ticket an. Ein Automat hält die Information darüber, wie viel Geld während seines Betriebs eingeworfen wurde. Reale Ticketautomaten bieten üblicherweise eine Auswahl an verschiedenen Tickets, von denen der Kunde eines auswählt. Unser Ticketautomat hat nur eine Ticketart mit einem Einheitspreis. Es ist um einiges anspruchsvoller, eine Klasse für einen Automaten zu schreiben, der verschiedene Ticketarten unterstützt. Aber immerhin können wir in der objektorientierten Programmierung mehrere Instanzen erzeugen, die jeweils einen eigenen Preis definieren, um so verschiedene Ticketarten zu unterstützen.

2.1.1 Das Verhalten des naiven Ticketautomaten

Konzept

Objekterzeugung: Einige Objekte können nur dann erzeugt werden, wenn zusätzliche Informationen bereitgestellt werden.

Öffnen Sie das Projekt *Naiver-Ticketautomat* in BlueJ. Dieses Projekt enthält nur eine Klasse – `Ticketautomat` –, die Sie ähnlich wie in Kapitel 1 untersuchen können. Wenn Sie eine Instanz von `Ticketautomat` erzeugen, werden Sie nach dem Preis gefragt, den ein Ticket des neuen Automaten kostet soll. Dieser Preis soll in Cent angegeben werden, also wäre eine Angabe wie 500 ein realistischer Wert.

Übung 2.1 Erzeugen Sie eine Instanz von `Ticketautomat` und untersuchen Sie ihre Methoden. Sie sollten folgende Methoden sehen: `geldEinwerfen`, `gibBisherGezahltenBetrag`, `gibPreis`, `ticketDrucken`. Rufen Sie die Methode `gibPreis` auf. Sie sollten einen Ergebniswert bekommen, der dem Wert entspricht, der bei der Erzeugung des Automaten angegeben wurde. Benutzen Sie die Methode `geldEinwerfen`, um das Einwerfen von Geld zu simulieren. Der Automat speichert den jeweiligen Stand nach jedem Münzeinwurf. Rufen Sie anschließend `gibBisherGezahltenBetrag` auf, um zu überprüfen, ob der Automat den eingeworfenen Betrag genau registriert hat. Sie können mehrere unterschiedliche Beträge einwerfen, genau wie an einem echten Automaten. Werfen Sie den exakten Betrag für ein Ticket ein und stellen Sie mit `gibBisherGezahltenBetrag` sicher, dass korrekt hochgezählt wurde. Da dies ein simpler Automat ist, wird das Ticket nicht automatisch ausgeworfen. Also rufen Sie die Methode `ticketDrucken` auf, sobald genügend Geld eingeworfen wurde. In der BlueJ-Konsole sollte nun der Ausdruck für ein Ticket erscheinen.

Übung 2.2 Welcher Wert wird geliefert, wenn Sie den bereits bezahlten Betrag ausgeben lassen, nachdem das Ticket gedruckt wurde?

Übung 2.3 Experimentieren Sie mit unterschiedlichen Beträgen, bevor Sie das Ticket drucken lassen. Verhält sich der Automat in irgendeiner Weise eigenartig? Was passiert, wenn Sie zu viel Geld einwerfen – bekommen Sie das Wechselgeld zurück? Was passiert, wenn Sie nicht genügend Geld einwerfen und dann versuchen, ein Ticket auszudrucken?

Übung 2.4 Versuchen Sie, das Verhalten des Automaten möglichst exakt zu verstehen, indem Sie mit der Instanz auf der Objektleiste interagieren, bevor wir uns im nächsten Abschnitt genauer ansehen, wie die Klasse `Ticketautomat` implementiert ist.

Übung 2.5 Erzeugen Sie einen weiteren Ticketautomaten, der einen anderen Preis verlangt. Denken Sie daran, dass Sie diesen Wert beim Erzeugen der Automaten-Instanz angeben müssen. Kaufen Sie an diesem Automaten ein Ticket. Sieht dieses Ticket anders aus als die Tickets, die von dem ersten Automaten ausgegeben werden?

2.2 Eine Klassendefinition untersuchen

Die Übungen am Ende des vorherigen Abschnitts zeigen, dass sich die Objekte der Klasse `Ticketautomat` nur dann sinnvoll verhalten, wenn wir exakt den geforderten Ticketpreis einwerfen. Wenn wir uns nun das Innenleben der Klasse ansehen, werden wir verstehen, warum das so ist.

Betrachten Sie den Quelltext der Klasse `Ticketautomat`, indem Sie einen Doppelklick auf deren Symbol im Klassendiagramm ausführen. Er sollte aussehen wie in Abbildung 2.1.

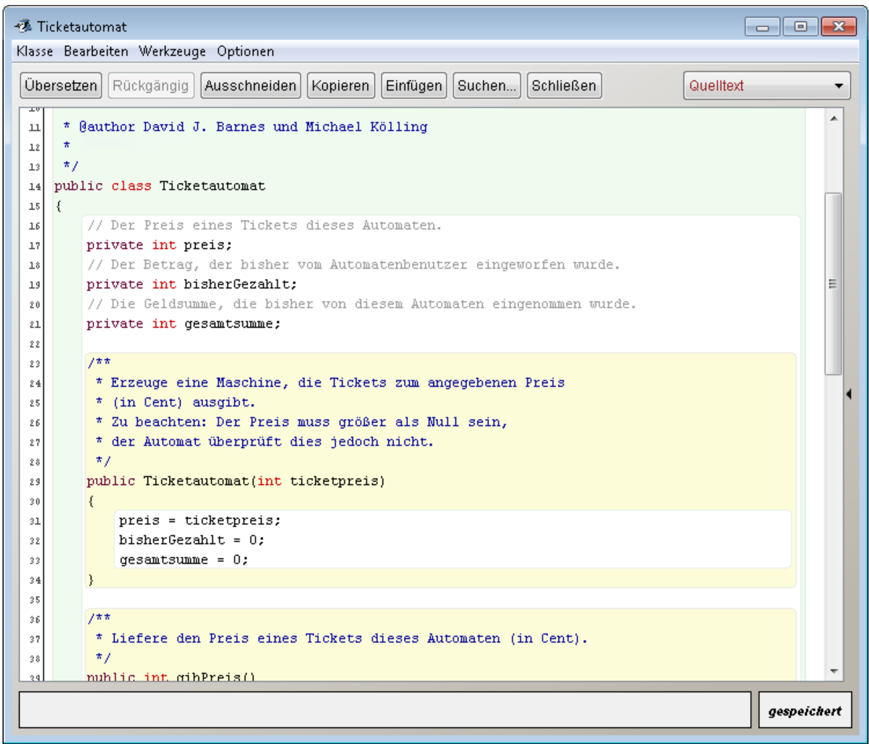


Abbildung 2.1 Das Editorfenster von BlueJ.

Der komplette Quelltext ist in Listing 2.1 zu sehen. Indem wir diesen Text Schritt für Schritt untersuchen, können wir die objektorientierten Konzepte herausschälen, die wir in Kapitel 1 bereits besprochen haben. Die im Quelltext definierte Klasse enthält etliche Elemente der Sprache Java, die uns wieder und wieder begegnen werden, sodass sich das gründliche Studium dieser Klasse auf jeden Fall auszahlt.

Listing 2.1
Die Klassendefinition der Klasse `Ticketautomat`.

```
/**
 * Die Klasse Ticketautomat modelliert einfache Ticketautomaten,
 * die Tickets zu einem Einheitspreis herausgeben.
 * Der Preis für die Tickets eines Automaten kann über den Konstruktor
 * festgelegt werden.
 * Ein Ticketautomat ist insofern 'naiv', dass er seinen Benutzern
 * vertraut, dass sie genügend Geld einwerfen, bevor sie sich ein Ticket
 * ausdrucken lassen.
 * Außerdem nimmt er an, dass sinnvolle Beträge eingeworfen werden.
 *
 * @author David J. Barnes und Michael Kölling
 * @version 2016.02.29
 */
public class Ticketautomat
{
    // Der Preis eines Tickets dieses Automaten.
    private int preis;
    // Der Betrag, der bisher vom Automatenbenutzer eingeworfen wurde.
    private int bisherGezahlt;
    // Die Geldsumme, die bisher von diesem Automaten eingenommen wurde.
    private int gesamtsumme;

    /**
     * Erzeuge eine Maschine, die Tickets zum angegebenen Preis
     * (in Cent) ausgibt.
     * Zu beachten: Der Preis muss größer als null sein,
     * der Automat überprüft dies jedoch nicht.
     */
    public Ticketautomat(int ticketpreis)
    {
        preis = ticketpreis;
        bisherGezahlt = 0;
        gesamtsumme = 0;
    }

    /**
     * Liefere den Preis eines Tickets dieses Automaten (in Cent).
     */
    public int gibPreis()
    {
        return preis;
    }

    /**
     * Liefere die Höhe des Betrags, der für das nächste Ticket bereits
     * eingeworfen wurde.
     */
    public int gibBisherGezahltenBetrag()
    {
        return bisherGezahlt;
    }

    /**
     * Nimm den angegebenen Betrag als Anzahlung für das
     * nächste Ticket.
     */
    public void geldEinwerfen(int betrag)
    {
        bisherGezahlt = bisherGezahlt + betrag;
    }
```

```
    /**
     * Drucke ein Ticket.
     * Aktualisiere die eingenommene Gesamtsumme und setze den gezahlten
     * Betrag auf null.
     */
    public void ticketDrucken()
    {
        // Den Ausdruck eines Tickets simulieren.
        System.out.println("##################");
        System.out.println("# Die BlueJ-Linie");
        System.out.println("# Ticket");
        System.out.println("# " + preis + " Cent.");
        System.out.println("##################");
        System.out.println();

        // Die Gesamtsumme mit dem eingezahlten Betrag aktualisieren.
        gesamtsumme = gesamtsumme + bisherGezahlt;
        // Die bisherige Bezahlung zurücksetzen.
        bisherGezahlt = 0;
    }
}
```

2.3 Der Kopf der Klasse

Der Quelltext einer Klasse kann in zwei Hauptbestandteile unterteilt werden: eine kleine äußere Klammer, die die Klasse benennt (und auf einem grünen Hintergrund dargestellt wird), und den meist viel umfangreicheren Innenteil, der die ganze Arbeit erledigt. In diesem Fall sieht die äußere Klammer folgendermaßen aus:

```
public class Ticketautomat
{
    Innenteil der Klasse hier ausgelassen
}
```

Die äußere Klammer sieht bei allen Klassen mehr oder weniger gleich aus; sie enthält den Kopf der Klasse, dessen Hauptzweck darin besteht, die Klasse zu benennen. Gemäß einer weitverbreiteten Konvention beginnen wir Klassennamen immer mit einem Großbuchstaben. Konsequent angewendet erlaubt uns diese Konvention, Klassennamen leicht von anderen Bezeichnungen zu unterscheiden – also z.B. von Variablen- oder Methodennamen, auf die wir gleich noch eingehen werden. Oberhalb des Kopfes der Klasse befindet sich ein Kommentar (in blauer Schrift), der uns Informationen über die Klasse gibt.

Übung 2.6 Schreiben Sie auf, wie die äußeren Klammern der Klassen `Student` und `Laborkurs` vermutlich aussehen – kümmern Sie sich nicht um den Innenteil.

Übung 2.7 Macht es einen Unterschiede, ob wir

```
public class Ticketautomat
```

oder

```
class public Ticketautomat
```

in der äußeren Klammer einer Klasse schreiben? Editieren Sie den Quelltext der Klasse `Ticketautomat` auf diese Weise und schließen Sie dann das Editorfenster. Stellen Sie einen Unterschied im Klassendiagramm fest?

Welche Fehlermeldung erhalten Sie, wenn Sie nun den Knopf ÜBERSETZEN klicken? Glauben Sie, dass diese Fehlermeldung klar erläutert, was falsch ist?

Nehmen Sie Ihre Änderung wieder zurück und achten Sie darauf, dass der Fehler nach dem Übersetzen nicht mehr auftaucht.

Übung 2.8 Überprüfen Sie, ob es möglich ist, das Wort `public` in der äußeren Klammer der Klasse `Ticketautomat` wegzulassen.

Übung 2.9 Fügen Sie das Wort `public` wieder ein und prüfen Sie dann, ob es möglich ist, das Wort `class` wegzulassen, bevor Sie den Code erneut übersetzen. Achten Sie darauf, beide Wörter wieder wie ursprünglich vorgegeben in den Quelltext einzutragen, bevor Sie mit der Lektüre dieses Kapitels fortfahren.

2.3.1 Schlüsselwörter

Die Wörter „public" und „class" sind Teil der Programmiersprache Java, während das Wort „Ticketautomat" ein bestimmter Name ist, der von dem Autor der Klasse frei gewählt wurde. Wir nennen Wörter wie „public" und „class" *Schlüsselwörter* oder *reservierte Wörter*. Diese Begriffe tauchen häufig auf und bedeuten das Gleiche. Es gibt in Java ungefähr 50 Schlüsselwörter und Sie werden schon bald die meisten davon kennen. Ein Punkt, auf den wir Sie noch hinweisen möchten, ist, dass Java-Schlüsselwörter niemals Großbuchstaben enthalten, während die Wörter, die wir selbst wählen (wie „Ticketautomat") oftmals eine Mischung aus Groß- und Kleinbuchstaben darstellen.

2.4 Datenfelder, Konstruktoren und Methoden

Im Innenteil einer Klasse werden die *Datenfelder*, *Konstruktoren* und *Methoden* definiert, die den Instanzen der Klasse ihre speziellen Eigenschaften und ihr Verhalten geben. Wir können die grundlegenden Aufgaben dieser drei Komponenten einer Klasse folgendermaßen zusammenfassen:

- Die Datenfelder speichern die Daten, die das jeweilige Objekt benutzt.
- Die Aufgabe der Konstruktoren ist es sicherzustellen, dass ein neu erzeugtes Objekt in einen vernünftigen Anfangszustand gesetzt wird.
- Die Methoden implementieren das Verhalten eines Objekts; sie liefern seine Funktionalität.

In BlueJ werden Datenfelder als Text auf einem weißen Hintergrund dargestellt, während Konstruktoren und Methoden innerhalb eines gelben Kastens dargestellt sind.

In Java gibt es nur wenige Regeln über die Reihenfolge, in der diese Elemente in einer Klassendefinition aufgeführt werden sollten. In der Klasse **Ticketautomat** haben wir uns entschieden, zuerst die Felder, dann die Konstruktoren und schließlich die Methoden aufzuführen (Listing 2.2). Diese Reihenfolge werden wir in allen unseren Beispielen einhalten. Andere Autoren bevorzugen eine andere Reihenfolge, und letztlich ist das eine Frage des Geschmacks. Unsere Konvention ist nicht unbedingt besser als andere. Dennoch ist es wichtig, sich auf eine Konvention zu einigen und diese dann konsequent durchzuhalten, weil Ihre Klassen dann einfacher zu lesen und zu verstehen sind.

```
public class Klassenname
{
    // Datenfelder
    // Konstruktoren
    // Methoden
}
```

Listing 2.2
Unsere Reihenfolge für Datenfelder, Konstruktoren und Methoden.

Übung 2.10 Durch das Experimentieren mit dem Ticketautomaten in BlueJ erinnern Sie sich vielleicht noch an die Namen einiger Methoden – beispielsweise **ticketDrucken**. Sehen Sie sich die Klassendefinition in Listing 2.3 an und benutzen Sie diese Information, zusammen mit den Hinweisen zur Reihenfolge, die wir oben gegeben haben, um eine Liste der Namen der Datenfelder, Konstruktoren und Methoden der Klasse **Ticketautomat** zu erstellen. *Hinweis*: Es gibt nur einen Konstruktor in der Klasse.

Übung 2.11 Welche zwei Eigenheiten des Konstruktors unterscheiden ihn signifikant von den Methoden einer Klasse?

2.4.1 Datenfelder

Datenfelder speichern Daten dauerhaft in einem Objekt. Die Klasse **Ticketautomat** hat drei Datenfelder: **preis**, **bisherGezahlt**, **gesamtsumme**. Diese Datenfelder werden auch als *Instanzvariablen* bezeichnet, da der Begriff *Variable* ein allgemeiner Ausdruck für Dinge ist, die Daten in einem Programm speichern. Wir haben sie gleich zu Anfang der Klassendefinition aufgeführt (Listing 2.3). Alle diese Variablen modellieren Geldbeträge, mit denen der Ticketautomat umgehen muss:

- **preis** ist ein Datenfeld für den festgelegten Preis eines Tickets.
- **bisherGezahlt** hält den Betrag, den der Benutzer bereits in den Automaten gesteckt hat, bevor er sich ein Ticket drucken lässt.
- **gesamtsumme** speichert die Summe aller Geldbeträge, die in den Automaten seit seiner Erzeugung von verschiedenen Benutzern eingeworfen wurden (ohne Berücksichtigung des aktuellen Betrags im Feld **bisherGezahlt**). Die Idee dahinter ist, dass nach Ausgabe eines Tickets der dafür gezahlte Betrag zu der Gesamtsumme hinzuaddiert wird.

Konzept

Datenfelder speichern die Daten, die ein Objekt benutzt. Datenfelder werden auch als Instanzvariablen bezeichnet.

Listing 2.3
Die Datenfelder der Klasse `Ticketautomat`.

```
public class Ticketautomat
{
    private int preis;
    private int bisherGezahlt;
    private int gesamtsumme;

    // Konstruktoren und Methoden hier ausgelassen
}
```

Datenfelder sind kleine Bereiche innerhalb eines Objekts, in denen Daten dauerhaft gespeichert werden können. Jedes Objekt reserviert bei seiner Erzeugung Platz für jedes Datenfeld, das in seiner Klasse deklariert wurde. Abbildung 2.2 zeigt schematisch eine Instanz der Klasse `Ticketautomat` mit ihren drei Datenfeldern. Den Datenfeldern sind noch keine Werte zugewiesen; sobald sie Werte haben, können wir den jeweiligen Wert an der Stelle eintragen, die das entsprechende Datenfeld repräsentiert. Die Darstellung hier ist ähnlich zu der eines Objekts auf der Objektleiste in BlueJ, enthält jedoch mehr Informationen. In BlueJ werden aus Platzgründen die Datenfelder bei den Objekten auf der Objektleiste nicht angezeigt. Wir können sie aber betrachten, wenn wir den Objektinspektor öffnen (Kapitel 1.7).

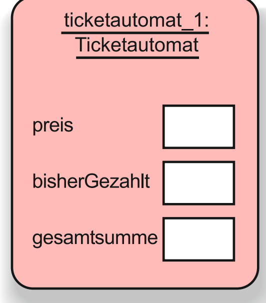


Abbildung 2.2
Ein Objekt der Klasse `Ticketautomat`.

Jedes Datenfeld hat seine eigene Deklaration im Quelltext. Vor jeder Deklaration haben wir eine Textzeile – einen Kommentar – eingefügt, der dem Leser den Zweck des Datenfelds beschreiben soll:

```
// Der Preis eines Tickets dieses Automaten.
private int preis;
```

Konzept

Kommentare werden im Quelltext einer Klasse angegeben, um menschlichen Lesern das Verstehen des Codes zu erleichtern. Kommentare haben keinen Einfluss auf die Funktionalität einer Klasse.

Ein einzeiliger Kommentar wird durch die beiden Zeichen // eingeleitet, zwischen denen kein Leerzeichen stehen darf. Ausführlichere Kommentare, die sich über mehrere Zeilen erstrecken, werden üblicherweise als Mehrzeilenkommentare formatiert. Diese beginnen mit dem Zeichenpaar /* und enden mit dem Paar */. Ein gutes Beispiel finden Sie vor dem Kopf der Klasse in Listing 2.1.

Die Definitionen der drei Datenfelder sind sehr ähnlich:

- Alle Definitionen besagen, dass sie *private* Datenfelder des Objekts sind; darauf werden wir in Kapitel 6 noch ausführlich eingehen, aber vorläufig belassen wir es bei der Regel, dass wir Datenfelder immer als privat vereinbaren.
- Alle drei Datenfelder sind vom Typ `int`. `int` ist ein weiteres Schlüsselwort, das den Datentyp Integer repräsentiert. Dies besagt, dass jedes Datenfeld einen ganzzahligen Wert speichern kann. Das ist sinnvoll, da wir ja Zahlen speichern wollen, die Geldbeträge in Cent darstellen.

Datenfelder können zu verschiedenen Zeitpunkten verschiedene Werte enthalten, deshalb werden sie auch *Variablen* genannt. Im Bedarfsfall können wir den Anfangswert, der in einem Datenfeld gespeichert ist, ändern. Wenn beispielsweise im Lauf der Zeit immer mehr Geld in den Ticketautomaten eingeworfen wird, dann werden wir den Wert im Datenfeld für die Gesamtsumme anpassen wollen. Es ist üblich, Datenfelder zu haben, deren Werte sich häufig ändern (wie **bisherGezahlt** und **gesamtsumme**), und solche, die sich nur selten oder gar nicht ändern (wie **preis**). Die Tatsache, dass sich der Wert von **preis** nach dem Setzen nicht mehr ändert, ändert nichts daran, dass es sich dennoch um eine Variable handelt. In den folgenden Abschnitten werden wir noch andere Arten von Variablen kennenlernen, die jedoch alle grundsätzlich den gleichen Zweck erfüllen, nämlich Daten zu speichern.

Die Datenfelder **preis**, **bisherGezahlt** und **gesamtsumme** sind Bestandteile eines Objekts der Klasse **Ticketautomat**, die es benötigt, um seine Aufgaben erfüllen zu können: Geld vom Kunden annehmen, Tickets drucken und eine Übersicht behalten, wie viel Geld bereits in den Automaten gezahlt wurde. In den nächsten Abschnitten werden wir untersuchen, wie diese Datenfelder vom Konstruktor und den Methoden benutzt werden, um das Verhalten von naiven Ticketautomaten zu implementieren.

Übung 2.12 Welchen Typ haben jeweils die folgenden Datenfelder Ihrer Meinung nach?

```
private int zaehler;
private Student sprecher;
private Server zentral;
```

Übung 2.13 Was sind die Namen der folgenden Datenfelder?

```
private boolean lebendig;
private Person tutor;
private Spiel spiel;
```

Übung 2.14 Beurteilen Sie, aufbauend auf Ihren Kenntnissen der Namenskonventionen für Klassen, welche der Typnamen in den ***Übungen 2.12*** und ***2.13*** Klassennamen sind.

Übung 2.15 Ist es bei der folgenden Deklaration eines Datenfeldes aus der Klasse **Ticketautomat**

```
private int preis;
```

wichtig, in welcher Reihenfolge die drei Wörter stehen? Editieren Sie die Klasse **Ticketautomat** und probieren Sie verschiedene Reihenfolgen aus. Gibt Ihnen das Aussehen des Klassendiagramms nach jeder Änderung einen Hinweis, ob andere Reihenfolgen möglich sind? Überprüfen Sie durch Klicken auf ÜBERSETZEN, ob es eine Fehlermeldung gibt.

Stellen Sie sicher, dass nach Ihren Experimenten die Originalversion wiederhergestellt wird!

Übung 2.16 Ist es zwingend erforderlich, dass am Ende einer Datenfelddeklaration ein Semikolon steht? Experimentieren Sie auch hier mit dem Editor. Die Regel, die Sie dabei lernen, ist sehr wichtig, deshalb sollten Sie sich diese einprägen.

Übung 2.17 Schreiben Sie die vollständige Deklaration eines Datenfeldes mit dem Typ `int` und dem Namen `status`.

Aus den Definitionen der Datenfelder, die wir bisher kennengelernt haben, schält sich ein Muster heraus, das sich auf jede Definition einer Datenfeldvariablen in einer Klasse anwenden lässt.

- Sie beginnen normalerweise mit dem reservierten Wort `private`.
- Sie enthalten einen Typnamen (wie `int`, `String`, `Person` usw.).
- Sie enthalten einen vom Benutzer gewählten Namen für die Datenfeldvariable.
- Sie enden mit einem Semikolon.

Merken Sie sich diese vier Punkte. Sie werden Ihnen später, wenn Sie Ihre eigenen Klassen schreiben, eine wertvolle Hilfe sein.

Wenn Sie sich die Mühe machen, den Quelltext verschiedener Klassen eingehender zu studieren, werden Sie immer wieder auf Muster wie diese stoßen. Teil des Lernprozesses beim Programmieren ist es, nach solchen Mustern Ausschau zu halten und sie dann in den eigenen Programmen zu verwenden. Das ist einer der Gründe, warum die genaue Analyse von Quelltext in diesem Stadium so nützlich ist.

2.4.2 Konstruktoren

Konzept

Konstruktoren ermöglichen, dass ein Objekt nach seiner Erzeugung in einen gültigen Zustand versetzt wird.

Konstruktoren haben eine ganz besondere Aufgabe: Sie sind verantwortlich dafür, dass ein Objekt unmittelbar nach seiner Erzeugung in einen gültigen Zustand versetzt wird, oder, um es anders auszudrücken, dass ein Objekt direkt nach seiner Erzeugung verwendet werden kann. Dieser Prozess wird auch als *Initialisierung* bezeichnet.

In gewisser Hinsicht kann ein Konstruktor mit einer Hebamme verglichen werden: Er ist dafür verantwortlich, dass das neue Objekt ordentlich ins Leben gerufen wird. Sobald ein Objekt erzeugt worden ist, spielt der Konstruktor im Leben des Objekts keine weitere Rolle mehr und kann auch nicht noch einmal aufgerufen werden. Listing 2.4 zeigt den Konstruktor der Klasse `Ticketautomat`.

Eines der besonderen Merkmale eines Konstruktors ist, dass er genauso heißt wie die Klasse, in der er definiert ist – in diesem Fall `Ticketautomat`. Der Name des Konstruktors folgt direkt, d.h. ohne etwas dazwischen, auf das Wort `public`.[1]

Erwartungsgemäß sollte es eine enge Verbindung geben zwischen dem, was im Rumpf eines Konstruktors geschieht und was in den Datenfelder der Klasse passiert.

1 Diese Behauptung vereinfacht geringfügig die entsprechende Java-Regel, ist aber für die Mehrzahl der Codebeispiele in diesem Buch gültig.

Der Grund ist, dass eine der Hauptaufgaben des Konstruktors darin besteht, die Datenfelder zu initialisieren. Einige Datenfelder, wie beispielsweise **bisherGezahlt** und **gesamtsumme**, können einen sinnvollen Wert bekommen, indem ihnen eine konstante Zahl, in diesem Fall die Null, zugewiesen wird. Mit anderen, wie hier dem Ticketpreis, ist das nicht ganz so einfach, weil wir den Preis eines Tickets nicht kennen, bevor wir den Automaten erzeugt haben: Erinnern Sie sich, dass wir möglicherweise mehrere Ticketautomaten mit unterschiedlichen Ticketpreisen haben möchten, sodass kein fester Preis für alle festgelegt werden kann. Beim Erzeugen von Ticketautomaten in BlueJ haben Sie ja festgestellt, dass Sie für jeden neuen Ticketautomaten den Preis der Tickets festlegen mussten. Ein wichtiger Punkt an dieser Stelle ist, dass der Preis eines Tickets an einer Stelle *außerhalb* des Ticketautomaten festgelegt wird und dass diese Information in den Automaten *hineingegeben* werden muss. In BlueJ entscheiden Sie sich für einen Wert und geben diesen in ein Dialogfeld ein. Eine Aufgabe des Konstruktors ist, diesen Wert entgegenzunehmen und ihn im Datenfeld **preis** des neu erzeugten Ticketautomaten abzulegen. Auf diese Weise kann der Automat den Wert behalten, ohne dass Sie ihn immer wieder daran erinnern müssen.

```
public class Ticketautomat
{
    //Datenfelder hier ausgelassen

    /**
     * Erzeuge einen Automaten, der Tickets zum angegebenen Preis
     * (in Cent) ausgibt.
     * Zu beachten: Der Preis muss größer als null sein,
     * der Automat überprüft dies jedoch nicht.
     */
    public Ticketautomat(int ticketpreis)
    {
        preis = ticketpreis;
        bisherGezahlt = 0;
        gesamtsumme = 0;
    }

    //Methoden hier ausgelassen
}
```

Listing 2.4 Der Konstruktor der Klasse **Ticketautomat**.

Wir sehen daran, dass eine der wichtigsten Aufgaben eines Datenfelds das Speichern von externen Informationen ist, damit diese Informationen dem Objekt über seine gesamte Lebensdauer zur Verfügung stehen können. Datenfelder bieten somit einen Ort, um Daten dauerhaft (d.h. persistent) zu speichern.

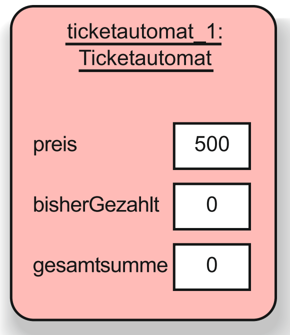


Abbildung 2.3 Eine Instanz der Klasse **Ticketautomat** nach ihrer Initialisierung (erzeugt mit einem Ticketpreis von 500 Cent).

Abbildung 2.3 zeigt eine Instanz der Klasse **Ticketautomat**, nachdem der Konstruktor ausgeführt wurde. Den Datenfeldern wurden nun Werte zugewiesen. Dem Diagramm können wir entnehmen, dass dieser Automat mit einem Ticketpreis von 500 erzeugt worden ist.

Im nächsten Abschnitt diskutieren wir, wie ein Objekt Werte von außen entgegennehmen kann.

Hinweis

In Java werden alle Datenfelder automatisch mit einem vordefinierten Wert belegt, falls sie nicht explizit gesetzt werden. Für Datenfelder vom Typ **int** ist dieser Standardwert 0. Genau genommen hätten wir also ohne die Initialisierung von **bisherGezahlt** und **gesamtsumme** mit 0 auskommen können, indem wir uns auf die Standardinitialisierung mit gleichem Ergebnis verlassen hätten. Wir bevorzugen aber die explizite Zuweisung. Sie hat keinen Nachteil, dokumentiert jedoch deutlicher, was tatsächlich passiert. Wir verlassen uns zum einen nicht darauf, dass der Leser die Standardwerte kennt. Zum anderen dokumentieren wir, dass die Datenfelder tatsächlich den Wert 0 bekommen sollen und wir nicht einfach vergessen haben, sie zu initialisieren.

2.5 Datenübergabe mit Parametern

Konstruktoren und Methoden spielen ganz unterschiedliche Rollen im Leben eines Objekts, aber die Art und Weise, wie beide Werte von außen erhalten, ist die gleiche: über *Parameter*. Vielleicht erinnern Sie sich daran, dass wir Parameter bereits kurz in Kapitel 1 (Abschnitt 1.4) kennengelernt haben. Parameter sind, wie Datenfelder, lediglich eine andere Form von Variablen und dienen somit der Aufnahme und Verwahrung von Daten. Parameter sind Variablen, die im Kopf eines Konstruktors oder einer Methode definiert werden:

```
public Ticketautomat(int ticketpreis)
```

Dieser Konstruktor hat einen einzelnen Parameter, **ticketpreis**, der vom Typ **int** ist – vom gleichen Typ wie das Datenfeld **preis**, dem dieser Wert zugewiesen werden soll. Ein Parameter wird als eine Art temporärer Bote verwendet, der Daten außerhalb des Konstruktors bzw. der Methode entgegennimmt und in den Konstruktor bzw. die Methode hineinreicht, sodass sie dort zur Verfügung stehen.

Abbildung 2.4 illustriert, wie Werte als Parameter übergeben werden. In diesem Fall gibt ein Benutzer von BlueJ einen Wert in ein Dialogfeld ein, wenn er einen Ticketautomaten erzeugt (links angedeutet). Dieser Wert wird in den Parameter **ticketpreis** des Konstruktors des neuen Automaten kopiert. Dies ist in der Abbildung durch den Pfeil (A) gekennzeichnet. Der Kasten in dem **Ticketautomat**-Objekt in Abbildung 2.4 mit der Aufschrift „Ticketautomat (Konstruktor)“ stellt zusätzlichen Platz dar, der nur während der Ausführung des Konstruktors zur Verfügung steht. Wir nennen ihn hier den *Konstruktorspeicher* des Objekts (oder *Methodenspeicher*, wenn wir über Methoden sprechen, denn bei diesen verhält es sich analog).

Der Konstruktorspeicher bietet Platz für die Werte der Konstruktorparameter. In unseren Diagrammen werden alle Variablen durch weiße Felder repräsentiert.

Wir unterscheiden zwischen den Parameter*namen* innerhalb eines Konstruktors (oder einer Methode) und den Parameter*werten* außerhalb, indem wir die Namen innerhalb *formale Parameter* und die Werte von außerhalb *aktuelle Parameter* nennen. Demnach ist beispielsweise `ticketpreis` ein formaler Parameter und ein Wert wie `500` ist ein aktueller Parameter.

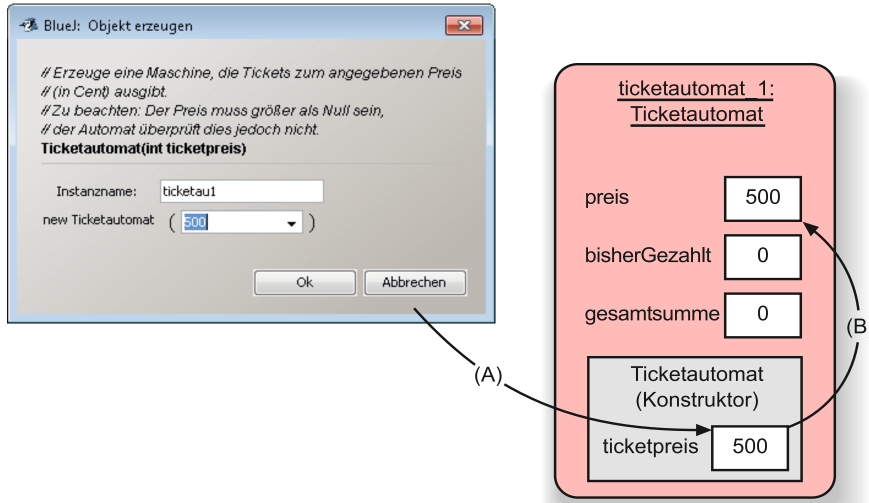


Abbildung 2.4
Parameterübergabe (A) und Zuweisung (B).

Ein formaler Parameter steht einem Objekt nur im Rumpf des Konstruktors oder der Methode zur Verfügung, in dem der Parameter deklariert ist. Wir sagen, dass die *Sichtbarkeit* eines Parameters auf den jeweiligen Rumpf beschränkt ist. Im Gegensatz dazu ist der Sichtbarkeitsbereich eines Datenfelds die gesamte Klassendefinition – es kann an allen Stellen in einer Klassendefinition benutzt werden. Dies ist ein sehr wichtiger Unterschied zwischen diesen beiden Arten von Variablen.

Konzept

Durch die **Sichtbarkeit** einer Variablen wird der Bereich im Quelltext definiert, von dem aus auf eine Variable zugegriffen werden kann.

Ein Konzept, das mit der Sichtbarkeit einer Variablen eng verwandt ist, ist das der *Lebensdauer*. Die Lebensdauer eines formalen Parameters ist auf die Ausführungszeit eines Konstruktors oder einer Methode beschränkt. Bei Aufruf eines Konstruktors oder einer Methode wird zusätzlicher Speicher für die Parametervariablen erzeugt und die externen Werte dort hinein kopiert. Nach der Ausführung verschwinden die formalen Parameter und ihre Werte gehen verloren. Mit anderen Worten: Nach Ausführung des Konstruktors wird der Konstruktorspeicher freigeräumt und mit ihm werden alle darin gehaltenen Parameterwerte entfernt (Abbildung 2.4).

Konzept

Die **Lebensdauer** einer Variablen legt fest, wie lange sie existiert, bevor sie zerstört wird.

Im Gegensatz dazu entspricht die Lebensdauer eines Datenfelds der des zugehörigen Objekts. Zusammen mit dem Objekt werden die Datenfelder erzeugt und sie bestehen, solange das Objekt besteht. Daher müssen wir, wenn wir uns den im Parameter `ticketpreis` gehaltenen Ticketpreis merken wollen, diesen an einem beständigeren Ort abspeichern – im Datenfeld `preis`.

Ebenso wie wir eine enge Verbindung zwischen einem Konstruktor und den Datenfeldern seiner Klasse erwarten, erwarten wir eine enge Verbindung zwischen den Parametern des Konstruktors und den Datenfeldern, da externe Werte oft benötigt werden, um die Werte von einem oder mehreren dieser Datenfelder zu setzen. Wo dies der Fall ist, stimmen die Parametertypen sehr eng mit den Typen der entsprechenden Datenfelder überein.

Übung 2.18 Zu welcher Klasse gehört der folgende Konstruktor?

```
public Student(String name)
```

Übung 2.19 Wie viele Parameter hat der folgende Konstruktor und welche Typen haben diese?

```
public Buch(String titel, double preis)
```

Übung 2.20 Können Sie die Typen der Datenfelder in der Klasse `Buch` erraten – und zwar aus den Parametern ihres Konstruktors? Können Sie etwas über die Namen dieser Datenfelder sagen?

2.5.1 Variablennamen wählen

Sicher ist Ihnen aufgefallen, dass die Variablennamen, die wir für Datenfelder und Parameter verwenden, eng mit dem Zweck der Variablen verbunden sind. Namen wie `preis`, `ticketpreis`, `titel` und `lebend` verraten Ihnen etwas über die Informationen, die in dieser Variablen gespeichert sind. Das macht es leichter zu verstehen, was im Programm abläuft. In Anbetracht der Tatsache, dass wir relativ frei in unserer Wahl des Variablennamens sind, lohnt es sich, das Prinzip zu beherzigen, Namen zu wählen, die den Zweck der Variablen kommunizieren, anstatt nur beliebig Buchstaben und Zahlen aneinanderzureihen.

2.6 Zuweisungen

Im vorigen Abschnitt haben wir angemerkt, dass der kurzlebige Wert einer Parametervariablen zu einer beständigeren Stelle kopiert werden sollte – zu einer Datenfeldvariablen. Der Rumpf des Konstruktors enthält deshalb die folgende *Zuweisung*:

```
preis = ticketpreis;
```

Konzept

Zuweisungen speichern den Wert auf der rechten Seite eines Zuweisungsoperators in der Variablen, die auf der linken Seite genannt ist.

Zuweisungen werden beim Programmieren häufig eingesetzt, um einen Wert in einer Variablen zu speichern. Sie sind an einem Zuweisungsoperator erkennbar, im obigen Beispiel `=`. Zuweisungen sind spezielle Anweisungen, die den Wert auf der rechten Seite des Operators nehmen und ihn in der Variablen speichern, die auf der linken Seite des Operators steht. In Abbildung 2.4 ist dies durch den Pfeil (B) gekennzeichnet. Die rechte Seite einer Zuweisung wird *Ausdruck* genannt. In ihrer allgemeinsten Form sind Ausdrücke Konstrukte, die Werte berechnen. In diesem Fall besteht der Ausdruck lediglich aus einer einzelnen Variablen, deren Wert in die Variable `preis` kopiert wird. Weiter hinten in diesem Kapitel werden wir noch kompliziertere Ausdrücke zu sehen bekommen.

Eine Regel für Zuweisungen ist, dass der Typ des Ausdrucks auf der rechten Seite zum Typ der Variablen auf der linken Seite passen muss, an die der Wert zugewiesen wird. Bisher haben wir nur drei verschiedene Typen kennengelernt: **int**, **String** und (nur sehr kurz) **boolean**. Die Regel besagt, dass wir beispielsweise keinen ganzzahligen Wert an eine **String**-Variable zuweisen können. Die gleiche Regel gilt auch zwischen einem formalen und einem aktuellen Parameter: Der Typ des aktuellen Parameters muss zum Typ der formalen Parametervariablen passen. Vorläufig können wir sagen, dass die Typen gleich sein müssen, obwohl wir in späteren Kapiteln sehen werden, dass dies nicht immer der Fall ist.

Übung 2.21 Nehmen Sie an, in einer Klasse **Haustier** gibt es ein Datenfeld **name** vom Typ **String**. Schreiben Sie eine Zuweisung in den Rumpf des folgenden Konstruktors, sodass das Datenfeld **name** mit dem Wert des Konstruktorparameters initialisiert wird.

```
public Haustier(String seinName)
{
}
```

Übung 2.22 *Zusatzaufgabe*. Die folgende Objekterzeugung führt dazu, dass der Konstruktor der Klasse **Datum** aufgerufen wird. Können Sie den Kopf des Konstruktors aufschreiben?

```
new Datum("Maerz", 23, 1861);
```

Versuchen Sie, den Parametern bedeutungsvolle Namen zu geben.

2.7 Methoden

Die Klasse **Ticketautomat** hat vier Methoden: **gibPreis**, **geldEinwerfen**, **gibBisherGezahltenBetrag** und **ticketDrucken**. Im Quelltext der Klasse (Listing 2.1) sehen Sie die Methoden auf gelbem Hintergrund dargestellt. Wir beginnen unsere Untersuchung des Quelltextes von Methoden, indem wir uns **gibPreis** näher ansehen (Listing 2.5).

```
public class Ticketautomat
{
    // Datenfelder hier ausgelassen

    // Konstruktor hier ausgelassen

    /**
     * Liefere den Preis eines Tickets dieses Automaten (in Cent).
     */
    public int gibPreis()
    {
        return preis;
    }

    // weitere Methoden hier ausgelassen
}
```

Listing 2.5
Die Methode **gibPreis**.

Konzept

Methoden bestehen aus zwei Teilen: einem Kopf und einem Rumpf.

Methoden bestehen aus zwei Teilen: einem *Kopf* und einem *Rumpf*. Der Kopf der Methode **gibPreis** einschließlich eines vorangestellten Kommentars sieht folgendermaßen aus:

```
/**
 * Liefere den Preis eines Tickets dieses Automaten (in Cent).
 */
public int gibPreis()
```

Es ist wichtig, zwischen dem Kopf einer Methode und der Deklaration eines Datenfelds zu unterscheiden, da sich beide sehr ähnlich sehen können. Dass **gibPreis** eine Methode und kein Datenfeld ist, können wir an dem Klammerpaar **()** erkennen, denn der Kopf einer Methode enthält immer ein solches Klammerpaar. Beachten Sie auch, dass am Ende des Kopfes kein Semikolon steht.

Der Rumpf der Methode ist alles das, was auf den Kopf folgt. Er ist immer in ein Paar geschweifte Klammern eingeschlossen: **{** und **}**. Der Rumpf einer Methode enthält *Deklarationen* und *Anweisungen*. Diese legen fest, was im Falle eines Aufrufs der Methode passiert. Deklarationen dienen dazu, zusätzlich temporären Speicherplatz für Variablen zu reservieren, während Anweisungen die Aktionen der Methoden beschreiben. In **gibPreis** enthält der Rumpf der Methode nur eine einzige Anweisung, aber wir werden bald Methodenrümpfe sehen, die aus etlichen Zeilen mit Deklarationen und Anweisungen bestehen.

Alle Deklarationen und Anweisungen, die zwischen geschweiften Klammern stehen, werden zusammengefasst als *Block* bezeichnet. Entsprechend ist der Rumpf der Klasse **Ticketautomat** ein Block und die Rümpfe des Konstruktors und der Methoden der Klasse sind ebenfalls Blöcke.

Es gibt mindestens zwei entscheidende Unterschiede zwischen dem Kopf des Konstruktors und dem der Methode **gibPreis**:

```
public Ticketautomat(int ticketpreis)
public int gibPreis()
```

- Die Methode hat einen *Ergebnistyp* vom Typ **int**, der Konstruktor hingegen hat keinen Ergebnistyp. Der Ergebnistyp wird unmittelbar vor den Methodennamen geschrieben.
- Der Konstruktor hat einen formalen Parameter, **ticketPreis**, die Methode hingegen hat keinen Parameter – nur ein Paar leere Klammern.

Es ist eine strikte Regel in Java, dass ein Konstruktor keinen Ergebnistyp definieren darf. Aber sowohl Konstruktoren als auch Methoden dürfen eine beliebige Anzahl an formalen Parametern definieren, also auch keine Parameter.

Im Rumpf von **gibPreis** steht eine einzelne Anweisung:

```
return preis;
```

Dies ist eine *Rückgabeanweisung*. Sie sorgt dafür, dass ein Wert vom Typ **int** zurückgeliefert wird, ganz so, wie es im Kopf der Methode definiert ist. Wenn eine Methode eine Rückgabeanweisung enthält, dann ist diese Anweisung immer die letzte in der Methode, denn nach Ausführung der Rückgabeanweisung ist auch die Ausführung einer Methode beendet.

Ergebnistypen und Rückgabeanweisungen bedingen einander. Der Ergebnistyp **int** von **gibPreis** ist eine Art Versprechen, dass der Rumpf der Methode etwas tut, das schließlich mit der Rückgabe eines **int**-Werts endet. Sie können sich den Aufruf einer Methode als eine Art Frage an ein Objekt vorstellen. Der Rückgabewert der Methode ist dann die Antwort auf die Frage. In diesem Fall lautet die Frage, wenn die Methode **gibPreis** an einem Ticketautomaten aufgerufen wird: „Was kostet ein Ticket?" Ein Ticketautomat muss für eine Antwort keine Berechnung durchführen, weil er die Antwort im Datenfeld **preis** hält. Entsprechend kann die Methode antworten, indem sie einfach den Wert dieser Variablen zurückliefert. Mit zunehmender Komplexität unserer Klassen werden wir unausweichlich auch auf kompliziertere Fragen stoßen, deren Beantwortung mehr Aufwand erfordert.

2.8 Sondierende und verändernde Methoden

Wir nennen Methoden wie die beiden **gib**-Methoden von **Ticketautomat** (**gibPreis** und **gibBisherGezahltenBetrag**) auch *sondierende Methoden*. Sie liefern dem Aufrufer Informationen über den Zustand des gerufenen Objekts – sie sondieren ihn. Eine sondierende Methode enthält üblicherweise eine Rückgabeanweisung, mit der ein bestimmter Wert zurückgeliefert wird.

Konzept

Sondierende Methoden liefern Informationen über den Zustand eines Objekts.

Es herrscht oft Verwirrung über das, was mit „als Wert zurückgeben" wirklich gemeint ist. Viele denken, dass damit das Programm angewiesen wird, etwas auszugeben. Dies trifft jedoch nicht zu – wir werden uns mit der Ausgabe beschäftigen, wenn wir uns die Methode **ticketDrucken** näher anschauen. Das Zurückgeben eines Werts bedeutet vielmehr, dass eine Information intern zwischen zwei verschiedenen Programmteilen weitergereicht wird. Ein Teil des Programms hat vermittels eines Methodenaufrufs Informationen von einem Objekt angefordert und über den Ergebniswert hat das Objekt die Möglichkeit, diese Information an den Aufrufer zurückzugeben.

Übung 2.23 Vergleichen Sie Kopf und Rumpf der Methode **gibBisherGezahltenBetrag** mit Kopf und Rumpf der Methode **gibPreis**. Welche Unterschiede bestehen zwischen den beiden?

Übung 2.24 Wenn ein Aufruf von **gibPreis** der Frage „Was kostet ein Ticket?" entspricht, welcher Frage entspricht dann ein Aufruf von **gibBisherGezahltenBetrag**?

Übung 2.25 Wenn der Name von **gibBisherGezahltenBetrag** in **gibBetrag** geändert würde, müsste dann auch die Rückgabeanweisung im Rumpf der Methode geändert werden? Probieren Sie es in BlueJ aus. Welche Schlüsse ziehen Sie daraus für den Namen einer sondierenden Methode und den Namen des damit verbundenen Datenfelds?

Übung 2.26 Schreiben Sie eine sondierende Methode **gibGesamtsumme** in der Klasse **Ticketautomat**. Die neue Methode sollte den Wert des Datenfelds **gesamtsumme** zurückliefern.