

A. Poetzsch-Heffter

Mitarbeit: J. Meyer, P. Müller

Aktualisiert: J. Knoop, M. Müller-Olm, U. Scheben, D. Keller, A. Thies

# Einführung in die objektorientierte Programmierung

mathematik  
und  
informatik



FernUniversität in Hagen

# Inhaltsverzeichnis

<b>Vorwort</b>	<b>VII</b>
<b>Studierhinweise zur Kurseinheit 1</b>	<b>1</b>
<b>1 Objektorientierung: Ein Einstieg</b>	<b>3</b>
1.1 Objektorientierung: Konzepte und Stärken . . . . .	3
1.1.1 Gedankliche Konzepte der Objektorientierung . . . . .	4
1.1.2 Objektorientierung als Antwort . . . . .	6
1.2 Paradigmen der Programmierung . . . . .	11
1.2.1 Prozedurale Programmierung . . . . .	12
1.2.2 Deklarative Programmierung . . . . .	15
1.2.3 Objektorientierte Programmierung: Das Grundmodell . .	18
1.3 Programmiersprachlicher Hintergrund . . . . .	24
1.3.1 Grundlegende Sprachmittel am Beispiel von Java . . . .	24
1.3.1.1 Objekte und Werte: Eine begriffliche Abgrenzung . . . . .	25
1.3.1.2 Objektreferenzen, Werte, Felder, Typen und Variablen . . . . .	26
1.3.1.3 Anweisungen, Blöcke und deren Ausführung . .	34
1.3.2 Objektorientierte Programmierung mit Java . . . . .	42
1.3.2.1 Objekte, Klassen, Methoden, Konstruktoren . .	42
1.3.2.2 Spezialisierung und Vererbung . . . . .	44
1.3.2.3 Subtyping und dynamisches Binden . . . . .	45
1.3.3 Aufbau eines Java-Programms . . . . .	46
1.3.4 Objektorientierte Sprachen im Überblick . . . . .	49
1.4 Aufbau und thematische Einordnung . . . . .	52
Selbsttestaufgaben zur Kurseinheit 1 . . . . .	55
Musterlösungen zu den Selbsttestaufgaben der Kurseinheit 1 . . . . .	61
<b>Studierhinweise zur Kurseinheit 2</b>	<b>71</b>
<b>2 Objekte, Klassen, Kapselung</b>	<b>73</b>
2.1 Objekte und Klassen . . . . .	73
2.1.1 Beschreibung von Objekten . . . . .	74

2.1.2	Klassen beschreiben Objekte . . . . .	75
2.1.3	Benutzen und Entwerfen von Klassen . . . . .	84
2.1.4	Weiterführende Sprachkonstrukte . . . . .	88
2.1.4.1	Initialisierung und Überladen . . . . .	88
2.1.4.2	Klassenmethoden und Klassenattribute . . . . .	93
2.1.4.3	Zusammenwirken der Spracherweiterungen . . . . .	95
2.1.5	Rekursive Klassendeklaration . . . . .	99
2.1.6	Typkonzept und Parametrisierung von Klassen . . . . .	100
2.1.6.1	Parametrisierung von Klassen . . . . .	103
2.1.6.2	Klassenattribute und -methoden und Überladen von Methodennamen im Kontext parametrischer Typen . . . . .	107
2.2	Kapselung und Strukturierung von Klassen . . . . .	110
2.2.1	Kapselung und Schnittstellenbildung: Erste Schritte . . . . .	111
2.2.2	Strukturieren von Klassen . . . . .	113
2.2.2.1	Innere Klassen . . . . .	113
2.2.2.2	Strukturierung von Programmen: Pakete . . . . .	123
2.2.3	Beziehungen zwischen Klassen . . . . .	131
	Selbsttestaufgaben zur Kurseinheit 2 . . . . .	135
	Musterlösungen zu den Selbsttestaufgaben der Kurseinheit 2 . . . . .	139
	<b>Studierhinweise zur Kurseinheit 3</b>	<b>145</b>
<b>3</b>	<b>Vererbung und Subtyping</b>	<b>147</b>
3.1	Klassifizieren von Objekten . . . . .	147
3.2	Subtyping und Schnittstellen . . . . .	156
3.2.1	Subtyping und Realisierung von Klassifikationen . . . . .	156
3.2.1.1	Deklaration von Schnittstellentypen und Subtyping . . . . .	157
3.2.1.2	Klassifikation und Subtyping . . . . .	162
3.2.1.3	Subtyping und dynamische Methodenauswahl . . . . .	167
3.2.2	Subtyping genauer betrachtet . . . . .	169
3.2.2.1	Subtyping bei vordefinierten Typen und Feldtypen . . . . .	169
3.2.2.2	Was es heißt, ein Subtyp zu sein . . . . .	173
3.2.3	Subtyping und Schnittstellen im Kontext parametrischer Typen . . . . .	176
3.2.3.1	Deklaration, Erweiterung und Implementierung parametrischer Schnittstellen . . . . .	176
3.2.3.2	Beschränkt parametrische Typen . . . . .	180
3.2.3.3	Subtyping bei parametrischen Behältertypen . . . . .	182
3.2.4	Typkonvertierungen und Typtests . . . . .	186
3.2.5	Unterschiedliche Arten von Polymorphie . . . . .	189
3.2.6	Programmieren mit Schnittstellen . . . . .	191

3.2.6.1	Die Schnittstellen <code>Iterable</code> , <code>Iterator</code> und <code>Comparable</code> . . . . .	192
3.2.6.2	Schnittstellen und Aufzählungstypen . . . . .	194
3.2.6.3	Schnittstellen zur Realisierung von Methodenparametern . . . . .	198
3.2.6.4	Beobachter und lokale Klassen . . . . .	201
	Selbsttestaufgaben zur Kurseinheit 3 . . . . .	207
	Musterlösungen zu den Selbsttestaufgaben der Kurseinheit 3 . . . . .	211
<b>Studierhinweise zur Kurseinheit 4</b>		<b>215</b>
3.3	Vererbung . . . . .	217
3.3.1	Vererbung: Das Sprachkonzept und seine Anwendung . . . . .	217
3.3.1.1	Vererbung von Programmteilen . . . . .	217
3.3.1.2	Erweitern und Anpassen von Ererbtem . . . . .	218
3.3.1.3	Spezialisieren mit Vererbung . . . . .	222
3.3.2	Vererbung, Subtyping und Subclassing . . . . .	228
3.3.3	Vererbung und Kapselung . . . . .	233
3.3.3.1	Kapselungskonstrukte im Zusammenhang mit Vererbung . . . . .	234
3.3.3.2	Zusammenspiel von Vererbung und Kapselung . . . . .	236
3.3.3.3	Realisierung gekapselter Objektgeflechte . . . . .	238
3.3.4	Verstecken von Attributen und Klassenmethoden vs. Überschreiben von Instanzmethoden . . . . .	249
3.3.5	Auflösen von Methodenaufrufen im Kontext überschriebener und überladener Methoden . . . . .	251
3.4	OO-Programmierung und Wiederverwendung . . . . .	254
<b>4 Bausteine für objektorientierte Programme</b>		<b>257</b>
4.1	Bausteine und Bibliotheken . . . . .	257
4.1.1	Bausteine in der Programmierung . . . . .	257
4.1.2	Überblick über die Java-Bibliothek . . . . .	261
4.2	Ausnahmebehandlung mit Bausteinen . . . . .	263
4.2.1	Eine Hierarchie von einfachen Bausteinen . . . . .	263
4.2.2	Zusammenspiel von Sprache und Bibliothek . . . . .	265
4.3	Stromklassen: Bausteine zur Ein- und Ausgabe . . . . .	268
4.3.1	Ströme: Eine Einführung . . . . .	268
4.3.2	Ein Baukasten mit Stromklassen . . . . .	272
4.3.2.1	Javas Stromklassen: Eine Übersicht . . . . .	273
4.3.2.2	Ströme von Objekten . . . . .	278
	Selbsttestaufgaben zur Kurseinheit 4 . . . . .	283
	Musterlösungen zu den Selbsttestaufgaben der Kurseinheit 4 . . . . .	289
<b>Studierhinweise zur Kurseinheit 5</b>		<b>299</b>

<b>5</b>	<b>Objektorientierte Programmgerüste</b>	<b>301</b>
5.1	Programmgerüste: Eine kurze Einführung . . . . .	302
5.2	Ein Gerüst für Bedienoberflächen: Das AWT . . . . .	305
5.2.1	Aufgaben und Aufbau graphischer Bedienoberflächen .	305
5.2.2	Die Struktur des Abstract Window Toolkit . . . . .	307
5.2.2.1	Das abstrakte GUI-Modell des AWT . . . . .	308
5.2.2.2	Komponenten . . . . .	309
5.2.2.3	Darstellung . . . . .	311
5.2.2.4	Ereignissteuerung . . . . .	314
5.2.2.5	Programmtechnische Realisierung des AWT im Überblick . . . . .	322
5.2.3	Praktische Einführung in das AWT . . . . .	323
5.2.3.1	Initialisieren und Anzeigen von Hauptfenstern	323
5.2.3.2	Behandeln von Ereignissen . . . . .	325
5.2.3.3	Elementare Komponenten . . . . .	328
5.2.3.4	Komponentendarstellung selbst bestimmen . .	332
5.2.3.5	Layout-Manager: Anordnen von Komponenten	335
5.2.3.6	Erweitern des AWT . . . . .	341
5.2.3.7	Rückblick auf die Einführung ins AWT . . . . .	345
5.3	Anwendung von Programmgerüsten . . . . .	346
5.3.1	Programmgerüste und Software-Architekturen . . . . .	346
5.3.2	Entwicklung graphischer Bedienoberflächen . . . . .	349
5.3.2.1	Anforderungen . . . . .	350
5.3.2.2	Entwicklung von Anwendungsschnittstelle und Dialog . . . . .	352
5.3.2.3	Entwicklung der Darstellung . . . . .	358
5.3.2.4	Realisierung der Steuerung . . . . .	361
5.3.2.5	Zusammenfassende Bemerkungen . . . . .	362
	Selbsttestaufgaben zur Kurseinheit 5 . . . . .	365
	Musterlösungen zu den Selbsttestaufgaben der Kurseinheit 5 . . . . .	369
	<b>Studierhinweise zur Kurseinheit 6</b>	<b>375</b>
<b>6</b>	<b>Parallelität</b>	<b>377</b>
6.1	Parallelität und Objektorientierung . . . . .	377
6.1.1	Allgemeine Aspekte von Parallelität . . . . .	378
6.1.2	Parallelität in objektorientierten Sprachen . . . . .	381
6.2	Lokale Parallelität in Java-Programmen . . . . .	382
6.2.1	Java-Threads . . . . .	382
6.2.1.1	Programmtechnische Realisierung von Threads in Java . . . . .	382
6.2.1.2	Benutzung von Threads . . . . .	387
6.2.2	Synchronisation . . . . .	397
6.2.2.1	Synchronisation: Problemquellen . . . . .	397

6.2.2.2	Ein objektorientiertes Monitorkonzept . . . . .	401
6.2.2.3	Synchronisation mit Monitoren . . . . .	406
6.2.3	Zusammenfassung der sprachlichen Umsetzung von lokaler Parallelität . . . . .	416
	Selbsttestaufgaben zur Kurseinheit 6 . . . . .	419
	Musterlösungen zu den Selbsttestaufgaben der Kurseinheit 6 . . . . .	425
<b>Studierhinweise zur Kurseinheit 7</b>		<b>433</b>
<b>7 Programmierung verteilter Objekte</b>		<b>435</b>
7.1	Verteilte objektorientierte Systeme . . . . .	435
7.1.1	Grundlegende Aspekte verteilter Systeme . . . . .	435
7.1.2	Programmierung verteilter objektorientierter Systeme . . . . .	438
7.2	Kommunikation über Sockets . . . . .	441
7.2.1	Sockets: Allgemeine Eigenschaften . . . . .	442
7.2.2	Realisierung eines einfachen Servers . . . . .	443
7.2.3	Realisierung eines einfachen Clients . . . . .	446
7.2.4	Client und Server im Internet . . . . .	447
7.2.4.1	Dienste im Internet . . . . .	449
7.2.4.2	Zugriff auf einen HTTP-Server . . . . .	452
7.2.4.3	Netzsurfer im Internet . . . . .	455
7.2.5	Server mit mehreren Ausführungssträngen . . . . .	457
7.3	Kommunikation über entfernten Methodenaufruf . . . . .	458
7.3.1	Problematik entfernter Methodenaufrufe . . . . .	459
7.3.1.1	Behandlung verteilter Objekte . . . . .	459
7.3.1.2	Simulation entfernter Methodenaufrufe über Sockets . . . . .	463
7.3.2	Realisierung von entfernten Methodenaufrufen in Java . . . . .	468
7.3.2.1	Der Stub-Skeleton-Mechanismus . . . . .	468
7.3.2.2	Entfernter Methodenaufruf in Java . . . . .	469
7.3.2.3	Parameterübergabe bei entferntem Metho- denaufruf . . . . .	475
<b>8 Zusammenfassung, Varianten, Ausblick</b>		<b>483</b>
8.1	Objektorientierte Konzepte zusammengefasst . . . . .	483
8.2	Varianten objektorientierter Sprachen . . . . .	486
8.2.1	Objektorientierte Erweiterung prozeduraler Sprachen . . . . .	486
8.2.2	Originär objektorientierte Sprachen . . . . .	490
8.2.2.1	Typisierte objektorientierte Sprachen . . . . .	490
8.2.2.2	Untypisierte objektorientierte Sprachen . . . . .	492
8.3	Zukünftige Entwicklungslinien . . . . .	494
	Selbsttestaufgaben zur Kurseinheit 7 . . . . .	497
	Musterlösungen zu den Selbsttestaufgaben der Kurseinheit 7 . . . . .	499

<b>Literaturverzeichnis</b>	<b>505</b>
<b>Stichwortverzeichnis</b>	<b>509</b>

# Vorwort

Die objektorientierte Programmierung modelliert und realisiert Software-Systeme als „Populationen“ kooperierender Objekte. Vom Prinzip her ist sie demnach eine Vorgehensweise, um Programme gemäß einem bestimmten Grundmodell zu entwerfen und zu strukturieren. In der Praxis ist sie allerdings eng mit dem Studium und der Verwendung objektorientierter Programmiersprachen verbunden: Zum einen gibt die programmiersprachliche Umsetzung den objektorientierten Konzepten konkrete und scharfe Konturen; zum anderen bilden die objektorientierten Sprachen eine unverzichtbare praktische Grundlage für die Implementierung objektorientierter Software.

Der Kurs stellt die konzeptionellen und programmiersprachlichen Aspekte so dar, dass sie sich gegenseitig befruchten. Jedes objektorientierte Konzept wird zunächst allgemein, d.h. unabhängig von einer Programmiersprache eingeführt. Anschließend wird ausführlich seine konkrete programmiersprachliche Umsetzung in Java erläutert. Zum Teil werden auch Realisierungsvarianten in anderen objektorientierten Sprachen vorgestellt. Aus praktischer Sicht ergibt sich damit insbesondere eine konzeptionell strukturierte Einführung in die Sprache und die Programmbibliothek von Java.

Inhaltlich ist der *Kurstext* in acht Kapitel eingeteilt. Die ersten drei Kapitel stellen die zentralen Konzepte und Sprachmittel vor. Kapitel 1 entwickelt das objektorientierte Grundmodell und vergleicht es mit den Modellen anderer Programmierparadigmen. Es fasst Grundbegriffe der Programmierung (insbesondere Variable, Wert, Typ, Ausdruck, Anweisung) zusammen und erläutert sie am Beispiel von Java. Anhand eines typischen Problems zeigt es Defizite der prozeduralen Programmierung auf und demonstriert, wie objektorientierte Techniken derartige Probleme bewältigen. Schließlich geht es nochmals genauer auf die Struktur des Kurses ein und ordnet ihn in verwandte und vertiefende Literatur ein. Kapitel 2 behandelt das Objekt- und Klassenkonzept und beschreibt, wie diese Konzepte in Java umgesetzt sind. Anschließend wird insbesondere auf Kapselungstechniken eingegangen. Kapitel 3 bietet eine detaillierte Einführung in Subtyping, Vererbung und Schnittstellenbildung und demonstriert, wie diese Konzepte in der Programmierung eingesetzt werden können und welche Schwierigkeiten mit ihrem Einsatz verbunden sind.

Die Kapitel 4 und 5 sind der Wiederverwendung von Klassen gewidmet.



Kapitel 4 geht zunächst auf solche Klassenhierarchien ein, deren Klassen nur in sehr eingeschränkter Form voneinander abhängen. Als Beispiele werden Bibliotheksklassen von Java herangezogen. Insbesondere wird auf die Verwendung von Stromklassen zur Ein- und Ausgabe eingegangen. Kapitel 5 behandelt eng kooperierende Klassen und sogenannte Programmgerüste (engl. Frameworks). Ausführlich wird in diesem Zusammenhang Javas Grundpaket zur Konstruktion graphischer Bedienoberflächen erläutert und seine Anwendung beschrieben.

In den Kapiteln 6 und 7 werden die spezifischen Aspekte der Realisierung paralleler und verteilter objektorientierter Programme behandelt. Kapitel 6 stellt dazu insbesondere das Thread-Konzept und die Synchronisationsmittel von Java vor. Kapitel 7 beschreibt die verteilte Programmierung mittels Sockets und entferntem Methodenaufruf. Schließlich bietet Kapitel 8 eine Zusammenfassung, behandelt Varianten bei der Realisierung objektorientierter Konzepte und gibt einen kurzen Ausblick.

**Java: warum, woher, welches?** Im Vergleich zu anderen objektorientierten Sprachen bietet die Abstützung und Konzentration auf Java wichtige Vorteile: Die meisten objektorientierten Konzepte lassen sich in Java relativ leicht realisieren. Java besitzt ein sauberes Typkonzept, was zum einen die Programmierung erleichtert und zum anderen eine gute Grundlage ist, um Subtyping zu behandeln. Java wird vermehrt in der industriellen Praxis eingesetzt und bietet außerdem eine hilfreiche Vorbereitung für das Erlernen der sehr verbreiteten, programmtechnisch aber komplexeren Sprache C++. Java ist eine Neuentwicklung und keine Erweiterung einer prozeduralen Sprache, so dass es relativ frei von Erblasten ist, die von den objektorientierten Aspekten ablenken (dies gilt z.B. nicht für Modula-3 und C++).

Ein weiterer Vorteil ist die freie Verfügbarkeit einfacher Entwicklungsumgebungen für Java-Programme und die umfangreiche, weitgehend standardisierte Klassenbibliothek. Aktuelle Informationen zu Java findet man auf der Web-Site der Firma Sun ausgehend von <http://java.sun.com>. Insbesondere kann man von dort die von Sun angebotene Software zur Entwicklung und Ausführung von Java-Programmen herunterladen oder bestellen. Die im vorliegenden Kurs beschriebenen Programme wurden mit den Versionen 1.2 und 1.3 des Java 2 Software Development Kits<sup>1</sup> (Standard Edition) entwickelt und getestet.

Hagen, Dezember 2001

*Arnd Poetzsch-Heffter*

---

<sup>1</sup>Java 2 SDK ist der neue Name für die früher als Java Development Kit, JDK, bezeichnete Entwicklungsumgebung.

Das Lehrgebiet Programmiersprachen und Softwarekonstruktion (Praktische Informatik V) wurde von Februar 2002 bis März 2003 von Dr. Jens Knoop und seit April 2003 von PD Dr. Markus Müller-Olm vertreten. Das unter Leitung von Professor Poetzsch-Heffter entwickelte Kursmaterial wurde für das Sommersemester 2003 von Dr. Jens Knoop und Ursula Scheben durchgesehen und an einigen Stellen ergänzt. Neben redaktionellen Nachbesserungen kamen dabei die folgenden Abschnitte neu zum Kurs hinzu: Der Unterabschnitt „Initialisierungsblöcke“ im Abschnitt 2.14, der Abschnitt 3.3.4 „Verstecken von Attributen und Klassenmethoden vs. Überschreiben von Instanzmethoden“ sowie der Abschnitt 3.3.5 „Auflösen von Methodenaufrufen im Kontext überschriebener und überladener Methoden“. Außerdem wurde am Ende jeder Kurseinheit ein Abschnitt mit Aufgaben und Wiederholungsfragen hinzugefügt. Für das Sommersemester 2004 wurde der Kurstext erneut durchgesehen und redaktionell überarbeitet.

Hagen, Dezember 2003

*Markus Müller-Olm*

Das ehemalige Lehrgebiet Programmiersprachen und Softwarekonstruktion wurde zum Lehrgebiet Programmiersysteme und wird seit Herbst 2004 von Professor Steimann geleitet. Das unter Leitung von Professor Poetzsch-Heffter entwickelte und von Professor Knoop und Professor Müller-Olm überarbeitete Kursmaterial wird für das Sommersemester 2007 von Dr. Ursula Scheben und Dr. Daniela Keller durchgesehen und erneut überarbeitet.

Der Kurstext wird u.a. an Java 5.0 angepasst. Dadurch ergeben sich an verschiedenen Stellen im Kurs Änderungen im Text und in den Programmbeispielen, u.a. werden Java Generics eingeführt. Die alten Übungsaufgaben, zu denen es keine Lösungsvorschläge gab, fallen weg und werden durch neue Beispiele mit Lösungen ersetzt. Desweiteren werden weniger relevante Abschnitte aus dem Kurs entfernt, andere erhalten mehr Gewicht. Entfernt wird z.B. das ehemalige Kapitel 2.1.7 „Klassen als Objekte“, während das bisherigen Kapitel 7 über verteilte, objektorientierte Systeme als grundlegend und wichtig eingestuft wird.

Hagen, Januar 2007

*Ursula Scheben*



# Studierhinweise zur Kurseinheit 1

Diese Kurseinheit beschäftigt sich mit dem ersten Kapitel des Kurstextes. Sie sollten dieses Kapitel im Detail studieren und verstehen. Ausgenommen von dem Anspruch eines detaillierten Verständnisses ist der Abschnitt 1.2.2 „Deklarative Programmierung“; bei dessen Inhalt reicht ein Verstehen der erläuterten Prinzipien. Schwerpunkte bilden die Abschnitte 1.1 „Objektorientierung: Konzepte und Stärken“ und der Abschnitt 1.3.1 „Grundlegende Sprachmittel am Beispiel von Java“. Nehmen Sie sich die Zeit, die Sprachkonstrukte an kleinen, selbst entworfenen Beispielen im Rahmen dieser Kurseinheit zu üben! Lassen Sie sich von den am Ende des Kapitels gegebenen Aufgaben zu eigenen selbständigen Übungen anregen. Ein bloßes Durchlesen dieses Kapitels ist nicht ausreichend.

Die Erfahrung lehrt, dass es sehr hilfreich ist, sich bei der Beschäftigung mit einem Kurs dessen inhaltliche Struktur genau zu vergegenwärtigen und die Kursstruktur als Einordnungshilfe und Gedächtnisstütze zu verwenden. Versuchen Sie deshalb, sich den in Abschnitt 1.4 „Aufbau und thematische Einordnung“ erläuterten Aufbau des Kurses einzuprägen.

## **Lernziele:**

- Grundlegende Konzepte der objektorientierten Programmierung.
- Basiskonzepte der objektorientierten Analyse.
- Unterschied zwischen objektorientierter Programmierung und anderen Programmierparadigmen.
- Sprachliche Grundlagen der Programmierung mit Java.
- Programmtechnische Fähigkeiten im Umgang mit Java.



# Kapitel 1

## Objektorientierung: Ein Einstieg

Dieses Kapitel hat zwei Schwerpunkte. Zum einen erläutert es den konzeptionellen Hintergrund der objektorientierten Programmierung und vergleicht ihn mit dem anderer Programmierparadigmen. Zum anderen vermittelt es die benötigten programmiersprachlichen Voraussetzungen. Insgesamt gibt es erste Antworten auf die folgenden Fragen:

1. Was sind die grundlegenden Konzepte und wo liegen die Stärken der objektorientierten Programmierung?
2. Wie unterscheidet sich objektorientierte Programmierung von anderen Programmierparadigmen?
3. Wie ist der Zusammenhang zwischen objektorientierten Konzepten und objektorientierten Programmiersprachen?

Jeder dieser Fragen ist ein Abschnitt gewidmet. Der dritte Abschnitt bietet darüber hinaus eine zusammenfassende Einführung in elementare Sprachkonzepte, wie sie in allen Programmiersprachen vorkommen, und erläutert ihre Umsetzung in Java. Abschnitt 1.4 beschreibt den Aufbau der folgenden Kapitel.

### 1.1 Objektorientierung: Konzepte und Stärken

Dieser Abschnitt bietet eine erste Einführung in objektorientierte Konzepte (Abschn. 1.1.1) und stellt die objektorientierte Programmierung als Antwort auf bestimmte softwaretechnische Anforderungen dar (Abschn. 1.1.2). Zunächst wollen wir allerdings kurz den Begriff „Objektorientierte Programmierung“ reflektieren. Dabei soll insbesondere deutlich werden, dass objektorientierte Programmierung mehr ist als die Programmierung in einer objektorientierten Programmiersprache.

Program-  
mierung

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

Obj.-or.  
Program-  
mierung

Objektorientierte Programmierung ist demnach Programmentwicklung mit Hilfe objektorientierter Konzepte und Techniken. Dabei spielen naturgemäß programmiersprachliche Aspekte eine zentrale Rolle. Resultat einer objektorientierten Programmentwicklung sind in der Regel, aber nicht notwendig, Programme, die in einer objektorientierten Programmiersprache verfasst sind. Im Gesamtbild der Softwareentwicklung wird die objektorientierte Programmierung durch objektorientierte Techniken für Analyse, Entwurf und Testen ergänzt.

### 1.1.1 Gedankliche Konzepte der Objektorientierung

Objekt

Die Objektorientierung bezieht ihre gedanklichen Grundlagen aus Vorgängen der realen Welt<sup>1</sup>. Vorgänge werden durch handelnde Individuen modelliert, die Aufträge erledigen und vergeben können. Dabei ist es zunächst unerheblich, ob die Individuen Personen, Institutionen, materielle Dinge oder abstrakte Gebilde sind. In der objektorientierten Programmierung werden die Individuen als *Objekte* bezeichnet. Dieser Abschnitt führt an einem kleinen Beispielszenario in die gedanklichen Konzepte der Objektorientierung ein und gibt eine erste Erläuterung der Begriffe „Nachricht“, „Methode“, „Klassifikation“ und „Vererbung“.

**Nachrichten und Methoden.** Ein zentraler Aspekt der Objektorientierung ist die Trennung von Auftragserteilung und Auftragsdurchführung. Betrachten wir dazu ein kleines Beispiel: Wir nehmen an, dass ein Herr P. ein Buch kaufen möchte. Um das Buch zu besorgen, erteilt Herr P. einem Buchhändler

<sup>1</sup>Der gedankliche Hintergrund der deklarativen Programmierung stammt aus der Mathematik, die prozedurale Programmierung hat sich durch Abstraktion aus der maschinennahen Programmierung und aus mathematischen Berechnungsverfahren entwickelt; vgl. Abschnitt 1.2.

den Auftrag, das Buch zu beschaffen und es ihm zuzuschicken. Die Auftragsdurchführung liegt dann in der Hand des Buchhändlers. Genauer besehen passiert Folgendes:

1. Herr P. löst eine Aktion aus, indem er dem Buchhändler einen Auftrag gibt. Übersetzt in die Sprache der Objektorientierung heißt das, dass ein Senderobjekt, nämlich Herr P., einem Empfängerobjekt, nämlich dem Buchhändler, eine *Nachricht* schickt. Diese Nachricht besteht üblicherweise aus der Bezeichnung des Auftrags (Buch beschaffen und zuschicken) und weiteren Parametern (etwa dem Buchtitel).
2. Der Buchhändler besitzt eine bestimmte *Methode*, wie er Herrn P.'s Auftrag durchführt (etwa: nachschauen, ob Buch am Lager, ansonsten billigsten Großhändler suchen, etc.). Diese Methode braucht Herr P. nicht zu kennen. Auch wird die Methode, wie das Buch beschafft wird, von Buchhändler zu Buchhändler im Allgemeinen verschieden sein.

*Nachricht*

*Methode*

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

**Klassifikation und Vererbung.** Die Klassifikation von Gegenständen und Begriffen durchzieht beinahe alle Bereiche unseres Lebens. Beispielsweise sind Händler und Geschäfte nach Branchen klassifiziert. Jede Branche ist dabei durch die Dienstleistungen charakterisiert, die ihre Händler erbringen: Buchhändler handeln mit Büchern, Lebensmittelhändler mit Lebensmitteln. Was im Geschäftsleben die Branchen sind, sind in der Sprache der Objektorientierung die Klassen. Eine *Klasse* legt die Methoden und Eigenschaften fest, die allen ihren Objekten gemeinsam sind. Klassen lassen sich hierarchisch organisieren. Abbildung 1.1 demonstriert eine solche hierarchische *Klassifikation* am Beispiel von Branchen. Dabei besitzen die übergeordneten Klassen nur

*Klasse*

*Klassifikation*

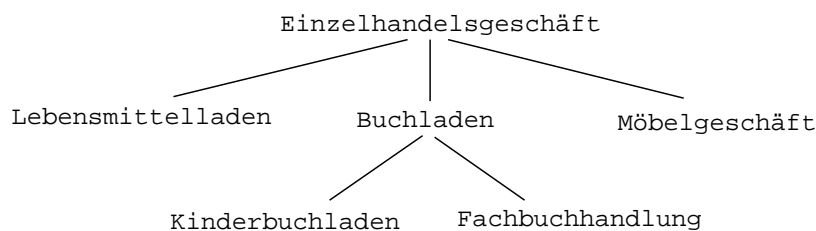


Abbildung 1.1: Klassifikation von Geschäften



Eigenschaften, die den untergeordneten Klassen bzw. ihren Objekten gemeinsam sind. Für die Beschreibung der Eigenschaften von Klassen und Objekten bringt die hierarchische Organisation zwei entscheidende Vorteile gegenüber einer unstrukturierten Menge von Klassen:

*abstrakte  
Klassen*

1. Es lassen sich *abstrakte* Klassen bilden; das sind Klassen, die nur dafür angelegt sind, Gemeinsamkeiten der untergeordneten Klassen zusammenzufassen. Jedes Objekt, das einer abstrakten Klasse zugerechnet wird, gehört auch zu einer der untergeordneten Klassen. Beispielsweise ist Einzelhandelsgeschäft eine abstrakte Klasse. Sie fasst die Eigenschaften zusammen, die allen Geschäften gemeinsam sind. Es gibt aber kein Geschäft, das nur ein Einzelhandelsgeschäft ist und keiner Branche zugeordnet werden kann. Anders ist es mit der Klasse Buchladen. Es gibt Buchläden, die zu keiner *spezielleren Klasse* gehören; d.h. die Klasse Buchladen ist nicht abstrakt.

*Vererbung*

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

*Spezialisierung*

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

Nach dieser kurzen Skizze der gedanklichen Konzepte, die der Objektorientierung zugrunde liegen, werden wir uns im folgenden Abschnitt den softwaretechnischen Anforderungen zuwenden, zu deren Bewältigung die objektorientierte Programmierung angetreten ist.

### 1.1.2 Objektorientierung als Antwort auf softwaretechnische Anforderungen

*Simula*

Objektorientierte Programmierung ist mittlerweile älter als 30 Jahre. Bereits die Programmiersprache *Simula 67* besaß alle wesentlichen Eigenschaften für die objektorientierte Programmierung (siehe z.B. [Lam88]). Die Erfolgsgeschichte der objektorientierten Programmierung hat allerdings erst Anfang der achtziger Jahre richtig an Fahrt gewonnen, stark getrieben von der

Programmiersprache *Smalltalk* (siehe [GR89]) und der zugehörigen Entwicklungsumgebung. Es dauerte nochmals ein Jahrzehnt, bis die objektorientierte Programmierung auch in der kommerziellen Programmentwicklung nennenswerte Bedeutung bekommen hat. Mittlerweile ist Objektorientierung so populär geworden, dass sich viele Software-Produkte, Werkzeuge und Vorgehensmodelle schon aus Marketing-Gründen objektorientiert nennen – unnötig zu sagen, dass nicht überall, wo „objektorientiert“ draufsteht, auch „objektorientiert“ drin ist.

*Small-  
talk*

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

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

1. softwaretechnische Simulation,
2. Konstruktion interaktiver, graphischer Bedienoberflächen,
3. Programm-Wiederverwendung und
4. verteilte Programmierung.

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

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

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

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

**2. Graphische Bedienoberflächen:** Interaktive, graphische Bedienoberflächen ermöglichen die nicht-sequentielle, interaktive Steuerung von Anwendungsprogrammen über direkt manipulierbare, graphische Bedienelemente wie Schaltflächen, Auswahlmenüs und Eingabefenster. Der Konstruktion graphischer Bedienoberflächen liegen eine ergonomische und zwei softwaretechnische Fragestellungen zugrunde:

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

### 3. Wiederverwendung von Programmen

Zwei Hauptprobleme stehen bei der Wiederverwendung von Programmen im Mittelpunkt:

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

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

### 4. Verteilte Programmierung

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

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

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

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

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

1. Sie verlangt ein inhärent paralleles Ausführungsmodell, mit dem insbesondere Bezüge zur realen Welt modelliert werden können. („Inhärent parallel“ bedeutet, dass Parallelität eine Eigenschaft des grundlegenden Ausführungsmodells ist und nicht erst nachträglich hinzugefügt werden muss.)
2. Die Strukturierung der Programme in kooperierende Programmteile mit klar definierten Schnittstellen spielt eine zentrale Rolle.
3. Anpassbarkeit, Klassifikation und Spezialisierung von Programmteilen ist eine wichtige Eigenschaft und sollte möglich sein, ohne bestehende Programmtexte manipulieren zu müssen.

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

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

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

begreift, spielen auch Aspekte der verteilten Programmierung eine wichtige Rolle für die Wiederverwendung.

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

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

## 1.2 Paradigmen der Programmierung

Eine Softwareentwicklerin, die ausschließlich funktionale Programme entwickelt hat, geht anders an eine softwaretechnische Problemstellung heran als ein Softwareentwickler, der nur mit Pascal gearbeitet hat. Sie benutzt andere Konzepte und Techniken, um Informationen zu organisieren und zu repräsentieren als ihr Berufskollege. Sie stützt ihre Entscheidungen auf andere Theorien und andere Standards. Engverzahnte Gebäude aus Konzepten, Vorgehensweisen, Techniken, Theorien und Standards fasst Thomas Kuhn (vgl. [Kuh76]) unter dem Begriff „Paradigma“ zusammen. Er benutzt den Begriff im Rahmen einer allgemeinen Untersuchung wissenschaftlichen Fortschritts. Wir verwenden diesen Begriff hier im übertragenen Sinne für den Bereich der Programmierung. Wie aus der Erläuterung zu ersehen ist, können sich unterschiedliche Paradigmen durchaus gegenseitig ergänzen. Dies gilt insbesondere auch für die Paradigmen der Programmierung.

*Paradigma*

Wir unterscheiden drei Programmierparadigmen: die prozedurale, deklarative und objektorientierte Programmierung. Dabei betrachten wir die prozedurale Programmierung als eine Erweiterung der imperativen Programmierung und begreifen die funktionale Programmierung als Spezialfall der deklarativen Programmierung. Um beurteilen zu können, was das Spezifi-

sche an der objektorientierten Programmierung ist und ob bzw. warum sie zur Lösung der im letzten Abschnitt erläuterten Aufgabenstellungen besonders geeignet ist, stellen wir in diesem Abschnitt die wesentlichen Eigenschaften alternativer Programmierparadigmen vor. Damit soll insbesondere auch in einer größeren Breite illustriert werden, was wir unter Programmierung verstehen, welche Konzepte, Techniken und Modelle dabei eine Rolle spielen und wie sie sich unterscheiden.

Eine zentrale Aufgabe der Softwareentwicklung besteht darin, allgemeine informationsverarbeitende Prozesse (z.B. Verwaltungsprozesse in Unternehmen, Steuerungsprozesse in Kraftanlagen oder Fahrzeugen, Berechnungsprozesse zur Lösung von Differentialgleichungen, Übersetzungsprozesse für Programme) so zu modellieren, dass sie von Rechenmaschinen verarbeitet werden können. Die Modellierung besteht im Wesentlichen aus der Modellierung der Informationen und der Modellierung der Verarbeitung.

Aufgabe der Programmierung ist es, die im Rahmen des Softwareentwurfs entwickelten Modelle soweit zu verfeinern, dass sie mittels Programmiersprachen formalisiert und damit auf Rechenmaschinen ausgeführt werden können. Die Programmierparadigmen unterscheiden sich dadurch, wie die Informationen und deren Verarbeitung modelliert werden und wie das Zusammenspiel von Informationen und Verarbeitung aussieht. Um die Vorstellung der unterschiedlichen Konzepte prägnant zu halten, werden wir ihre Darstellung angemessen vereinfachen. In der Praxis finden sich selbstverständlich Mischformen dieser Konzepte. Modulkonzepte lassen wir zunächst unberücksichtigt, da sie für alle Paradigmen existieren und deshalb zur Unterscheidung nicht wesentlich beitragen.

Die folgenden Abschnitte erläutern, wie Informationen und deren Verarbeitung in den unterschiedlichen Programmierparadigmen modelliert und beschrieben werden.

### 1.2.1 Prozedurale Programmierung

In der prozeduralen Programmierung ist die Modellierung der Informationen von der Modellierung der Verarbeitung klar getrennt. Informationen werden im Wesentlichen durch Grunddaten (ganze Zahlen, boolesche Werte, Zeichen, Zeichenreihen usw.) modelliert, die in Variablen gespeichert werden. Variablen lassen sich üblicherweise zu Feldern (Arrays) oder Verbunden (Records) organisieren, um komplexere Datenstrukturen zu realisieren. Darüber hinaus kann man mit Referenzen/Zeigern rechnen, die auf Variable verweisen.

Die Verarbeitung wird modelliert als eine Folge von globalen Zustandsübergängen, wobei sich der globale *Zustand* aus den Zuständen der Variablen zusammensetzt (in der Praxis gehören auch die vom Programm bearbeiteten Dateien zum Zustand). Bei jedem Zustandsübergang wird der Inhalt einer oder einiger weniger Variablen verändert. Möglicherweise können bei einem

*Zustand*

Zustandsübergang auch Variablen erzeugt oder entfernt werden. Ein Zustandsübergang wird durch eine elementare Anweisung beschrieben (z.B. in Pascal durch eine Zuweisung, `x := 4`; eine Allokation, `new(p)`; eine Deallokation, `dispose(p)`, oder eine Lese- bzw. Schreiboperation); d.h. es wird explizit vorgeschrieben, wie der Zustand zu ändern ist (daher auch der Name *imperative Programmierung*). Folgen von Zustandsübergängen werden durch zusammengesetzte Anweisungen<sup>2</sup> beschrieben. Zusammengesetzte Anweisungen können auch unendliche, d.h. nichtterminierende Übergangsfolgen beschreiben; dies ist zum Beispiel wichtig zur Programmierung von Systemen, die bis zu einem unbestimmten Zeitpunkt ununterbrochen Dienste anbieten sollen, wie z.B. Betriebssysteme, Netzserver oder die Event-Behandlung von graphischen Bedienoberflächen.

Eine *Prozedur* ist eine benannte, parametrisierte Anweisung, die meistens zur Erledigung einer bestimmten Aufgabe bzw. Teilaufgabe dient; Prozeduren, die Ergebnisse liefern (zusätzlich zur möglichen Veränderung, Erzeugung oder Entfernung von Variablen), werden *Funktionsprozeduren* genannt. Eine Prozedur kann zur Erledigung ihrer Aufgaben andere Prozeduren aufrufen. Jeder Programmstelle mit einem Prozeduraufruf ist eindeutig eine auszuführende Prozedur zugeordnet;<sup>3</sup> z.B. wird im folgenden Programmfragment in Zeile (2) bei jedem Schleifendurchlauf die Prozedur `proz007` aufgerufen:

Prozedur

```
(1) while( not abbruchbedingung ) {
    ...
(2)     proz007(...);
    ...
}
```

Wie wir sehen werden, gibt es in der objektorientierten Programmierung keine derartige eindeutige Zuordnung von Aufrufstelle zu Prozedur.



Das Grundmodell der Verarbeitung in prozeduralen Programmen ist die Zustandsänderung. Bei terminierenden Programmen wird ein Eingabezustand in einen Ausgabezustand transformiert, bei nichtterminierenden Programmen wird ein initialer Zustand schrittweise verändert. Die gesamte Zustandsänderung wird in einzelne Teiländerungen zerlegt, die von Prozeduren ausgeführt werden:

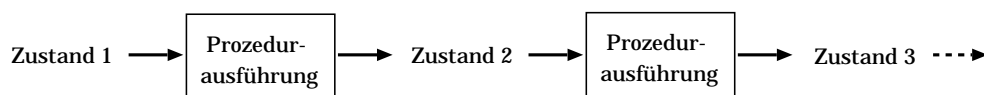


Abbildung 1.2: Schrittweise Änderung globaler Zustände

<sup>2</sup>Z.B. bedingte Anweisung, Schleifen etc.

<sup>3</sup>Wir sehen an dieser Stelle von Prozedurparametern und Prozedurvariablen ab.



Damit lässt sich das Grundmodell der prozeduralen Programmierung so zusammenfassen: Modelliere die Informationen durch (im Wesentlichen globale) Zustände über Variablen und Daten und modelliere die Verarbeitung als schrittweise Änderung des Zustands. Zustandsänderungen werden durch Anweisungen beschrieben. Mehrfach verwendete Anweisungsfolgen lassen sich zu Prozeduren zusammenfassen. Prozeduren bilden demnach das zentrale Strukturierungsmittel prozeduraler Programme. Sie strukturieren aber nur die Verarbeitung und nicht die Modellierung der Information.

**Frage.** Was sind die Schwächen der prozeduralen Programmierung in Hinblick auf die drei in Abschn. 1.1.2 skizzierten Anforderungen?

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

Das Grundmodell der prozeduralen Programmierung erlaubt in natürlicher Weise die Strukturierung der Verarbeitung. Es bietet aber wenig Möglichkeiten, um Teile des Zustands mit den auf ihm operierenden Prozeduren zusammenzufassen und die Kapselung von Daten zu erreichen. Dieser Schwäche wird mit Modulkonzepten begegnet. Die üblichen Modulkonzepte gestatten es, Typen, Variablen und Prozeduren zusammenzufassen, also die Programmtexte zu strukturieren. Mit Modulen kann man aber nicht rechnen: Sie können nicht als Parameter beim Prozeduraufruf übergeben oder Variablen zugewiesen werden; insbesondere ist es normalerweise nicht möglich, während der Programmausführung Kopien von Modulen zu erzeugen.

Prozedurale Programmierung wird meist im Zusammenhang mit strenger Typisierung behandelt. Typkonzepte verbessern zwar die statische Überprüfbarkeit von Programmen, erschweren aber deren Anpassbarkeit. Die Typen der Parameter einer Prozedur  $p$  sind fest vorgegeben. Wird ein Typ erweitert oder modifiziert, entsteht ein neuer Typ, dessen Elemente von  $p$  nicht mehr akzeptiert werden, selbst wenn die Änderungen keinen Einfluss auf die Bearbeitung hätten. In diesem Bereich hat man durch Prozedurparameter, generische Prozeduren und generische Module Verbesserungen erzielt.

Die prozedurale Programmierung ermöglicht eine Modellierung informationsverarbeitender Prozesse mit relativ einfachen, effizient zu implementierenden Mitteln. Diese Einfachheit ist ihre Stärke. Ihr Grundmodell bedarf aber einiger Erweiterungen, um den im letzten Abschnitt skizzierten Anforderungen gerecht zu werden. Aus der Summe dieser Erweiterungen resultiert dann allerdings ein komplexes Programmiermodell. Eine sprachliche Umsetzung führt zu sehr umfangreichen Programmiersprachen (typisches Beispiel hierfür ist die Sprache Ada).

## 1.2.2 Deklarative Programmierung

Die deklarative Programmierung hat das Ziel, mathematische Beschreibungsmittel für die Programmierung nutzbar zu machen. Damit sollen vor allem zwei Schwächen der imperativen bzw. prozeduralen Programmierung überwunden werden. Zum einen soll der Umgang mit komplexeren Daten wie Listen, Bäumen, Funktionen und Relationen erleichtert werden, zum anderen soll das oft fehleranfällige Arbeiten mit Variablen überwunden werden. Die deklarative Programmierung verzichtet im Wesentlichen auf den Zustandsbegriff. Ein deklaratives Programm ist nicht mehr eine Verarbeitungsvorschrift, sondern eine Spezifikation der gewünschten Programmergebnisse mit speziellen mathematischen Beschreibungsmitteln. Im Mittelpunkt steht also die Modellierung von Informationen, ihren Beziehungen und Eigenschaften. Die Verarbeitung der Informationen geschieht in deklarativen Programmiermodellen zum Großteil implizit.

Die deklarative Programmierung kennt mehrere Ausprägungen, die sich im Wesentlichen durch die verwendeten mathematischen Beschreibungsmittel unterscheiden. Wir stellen hier die funktionale und logische Programmierung kurz vor, um die obigen allgemeinen Ausführungen zu illustrieren.

**Funktionale Programmierung.** Die funktionale Programmierung betrachtet ein Programm als eine partielle Funktion von Eingabe- auf Ausgabedaten. Die Ausführung eines funktionalen Programms entspricht der Anwendung der Funktion auf eine Eingabe.

Ein funktionales Programm besteht im Wesentlichen aus Deklarationen von Datentypen und Funktionen, wobei Parameter und Ergebnisse von Funktionen selbst wieder Funktionen sein können (solche Funktionen nennt man *Funktionen höherer Ordnung*). Da funktionale Programmierung keine Variablen, keine Zeiger und keine Schleifen kennt, kommt der Rekursion zur Definition von Datenstrukturen und Funktionen eine zentrale Rolle zu. Mit einem kleinen funktionalen Programm, das prüft, ob ein Binärbaum Unterbaum eines anderen Binärbaums ist, wollen wir diese Art der Programmierung kurz illustrieren.

Ein Binärbaum ist entweder ein Blatt oder eine Astgabel mit zwei Teilbäumen. Ein Binärbaum *a* ist Unterbaum von einem Blatt, wenn *a* ein Blatt ist; ein Binärbaum ist Unterbaum von einem zusammengesetzten Binärbaum mit Teilbäumen *b1* und *b2*, wenn er (1) gleich dem zusammengesetzten Binärbaum ist oder (2) Unterbaum von *b1* ist oder (3) Unterbaum von *b2* ist. Formuliert in der Syntax der funktionalen Programmiersprache Gofer ergibt sich aus dieser Definition folgendes kompakte Programm:

```
data  BinBaum = Blatt  |  AstGabel  BinBaum BinBaum

istUnterbaum :: BinBaum -> BinBaum -> Bool
```

```

istUnterbaum a Blatt    = ( a == Blatt )
istUnterbaum a (AstGabel b1 b2) =
                                ( a == (AstGabel b1 b2) )
                                || ( istUnterbaum a b1 )
                                || ( istUnterbaum a b2 )

```

Das Schlüsselwort `data` leitet eine Datentypdeklaration ein, hier die Deklaration des Typs `BinBaum` mit den beiden Alternativen `Blatt` und `AstGabel`. Die Funktion `istUnterbaum` nimmt zwei Werte vom Typ `BinBaum` als Eingabe und liefert einen booleschen Wert als Ergebnis; `istUnterbaum` angewendet auf einen Binärbaum `a` und ein `Blatt` liefert `true`, wenn `a` ein `Blatt` ist; `istUnterbaum` angewendet auf eine `AstGabel` liefert `true`, wenn einer der drei angegebenen Fälle erfüllt ist.

Die funktionale Programmierung ermöglicht es also insbesondere, komplexe Datenstrukturen (im Beispiel Binärbäume) direkt, d.h. ohne Repräsentation mittels verzeigerten Variablen, zu deklarieren und zu benutzen. Besondere Stärken der funktionalen Programmierung sind das Programmieren mit Funktionen höherer Ordnung, ausgefeilte, parametrische Typsysteme (parametrischer Polymorphismus) und flexible Modularisierungskonzepte.

**Logische Programmierung.** Die logische Programmierung betrachtet ein Programm als eine Ansammlung von Fakten und Folgerungsbeziehungen, an die Anfragen gestellt werden können. Die Ausführung eines logischen Programms sucht Antworten auf solche Anfragen.

Die logische Programmierung bedient sich einer Sprache der formalen Logik, um Fakten und ihre Zusammenhänge zu beschreiben. Beispielsweise könnten wir die Tatsache, dass Sokrates ein Mensch ist, dadurch ausdrücken, dass ein Prädikat `istMensch` für Sokrates gilt. Die Aussage, dass alle Menschen sterblich sind, könnten wir als Folgerung formulieren, indem wir postulieren, dass jeder, der das Prädikat `istMensch` erfüllt, auch das Prädikat `sterblich` erfüllen soll. In der Syntax der Programmiersprache PROLOG erhält man damit folgendes Programm:

```

istMensch( sokrates ).
sterblich(X) :- istMensch(X).

```

Die Ausführung solcher Programme wird über Anfragen ausgelöst. Auf die Anfrage `sterblich( sokrates )?` würde die Programmausführung mit „ja“ antworten. Auf eine Anfrage `sterblich( X )?` sucht die Programmausführung nach allen Termen, die das Prädikat `sterblich` erfüllen. Auf der Basis unseres Programms kann sie dies nur für den Term `sokrates` ableiten. Eine Anfrage `sterblich( kohl )?` würde mit „nein“ beantwortet werden, da die Sterblichkeit von Kohl aus den angegebenen Fakten nicht abgeleitet werden kann.

Zwei weitere Ausprägungen der deklarativen Programmierung seien zumindest erwähnt: die Programmierung mit Constraints und die relationale Programmierung im Zusammenhang mit Datenbanken. Constraint-Programmierung kann als eine Verallgemeinerung der logischen Programmierung begriffen werden, bei der der Zusammenhang von Daten nicht nur mittels Folgebeziehungen formuliert werden kann. In der relationalen Datenbankprogrammierung werden Fakten mittels endlicher Relationen formuliert und gespeichert. Eine Datenbank ist im Wesentlichen eine Menge solcher Relationen (formal gesehen also eine Variable vom Typ „Menge von Relationen“). Mit Abfragesprachen, die auf den Relationenkalkül abgestützt sind, lässt sich der Inhalt von Datenbanken abfragen und modifizieren. Die relationale Datenbankprogrammierung ist sicherlich die ökonomisch mit Abstand bedeutendste Variante der deklarativen Programmierung.

Ein Schwerpunkt der Forschung im Bereich deklarativer Programmierung zielt auf die Integration der unterschiedlichen Formen deklarativer Programmierung ab. So wurde die logische Programmierung zum allgemeinen Constraint-Lösen erweitert, es wurden mehrere Ansätze erarbeitet, funktionale und logische Programmierung zu kombinieren, und es wurden sogenannte deduktive Datenbanken entwickelt, die Techniken der logischen Programmierung für relationale Datenbanken nutzbar machen. Dabei gestaltet sich die Integration der mathematischen Beschreibungskonzepte meist relativ einfach. Probleme macht die Integration der oft impliziten und recht unterschiedlichen Ausführungsmodelle und ihrer Implementierungen.

**Frage.** Was sind die Schwächen der deklarativen Programmierung in Hinblick auf die drei in Abschn. 1.1.2 skizzierten Anforderungen?

In der deklarativen Programmierung spielen die Ausführungsmodelle eine untergeordnete Rolle. Deshalb ist das Grundmodell der deklarativen Programmierung wenig geeignet, parallele Prozesse der realen Welt zu modellieren, bei denen räumlich verteilte Objekte eine Rolle spielen, die im Laufe der Ausführung erzeugt werden, in Beziehung zu anderen Objekten treten, ihren Zustand ändern und wieder verschwinden. Modellierung und Realisierung von verteilten Prozessen wird in der deklarativen Programmierung vielfach durch spezielle Konstrukte erreicht, zum Beispiel durch Einführung expliziter Kommunikationskanäle, über die Ströme von Daten ausgetauscht werden können. Ähnliches gilt für die Beschreibung nichtterminierender Prozesse oder zeitlich verzahnter Interaktionen zwischen Programmteilen bzw. zwischen dem Programm und dem Benutzer.

Die deklarative Programmierung eignet sich gut für eine hierarchische Strukturierung von Programmen. Für eine Strukturierung in kooperierende Programmteile müssen zwei Nachteile überwunden werden:

1. Deklarative Programmierung geht implizit davon aus, alle Daten „zen-

tral“ verfügbar zu haben, und bietet wenig Hilfestellung, Daten auf Programmteile zu verteilen.

2. Da Programmteile nicht über die Änderung an Datenstrukturen kommunizieren können, müssen tendenziell mehr Daten ausgetauscht werden als in imperativen Programmiermodellen.

In der Literatur geht man davon aus, dass sich deklarative Programme leichter modifizieren lassen als prozedurale Programme. Zwei Argumente sprechen für diese These: Deklarative Programmierung vermeidet von vornherein einige Fehlerquellen der prozeduralen Programmierung (Seiteneffekte, Zeigerprogrammierung, Speicherverwaltung); Schnittstelleneigenschaften deklarativer Programme lassen sich einfacher beschreiben. Andererseits gelten die Anmerkungen zur Anpassbarkeit prozeduraler Programme entsprechend auch für deklarative Programme.

Die deklarative Programmierung stellt für spezifische softwaretechnische Aufgabenstellungen (z.B. Datenbanken) sehr mächtige Programmierkonzepte, -techniken und -werkzeuge zur Verfügung. Sie ermöglicht in vielen Fällen eine sehr kompakte Formulierung von Programmen und eignet sich durch ihre Nähe zu mathematischen Beschreibungsmitteln gut für die Programmentwicklung aus formalen Spezifikationen.

### 1.2.3 Objektorientierte Programmierung: Das Grundmodell

*Objekte haben  
einen Zustand*

*Objekte  
bearbeiten  
Nachrichten*

Die objektorientierte Programmierung betrachtet eine Programmausführung als ein System kooperierender Objekte. Objekte haben einen eigenen lokalen Zustand. Sie haben eine gewisse Lebensdauer, d.h. sie existieren vor der Programmausführung oder werden während der Programmausführung erzeugt und leben, bis sie gelöscht werden bzw. bis die Programmausführung endet. Objekte empfangen und bearbeiten Nachrichten. Bei der Bearbeitung von Nachrichten kann ein Objekt seinen Zustand ändern, Nachrichten an andere Objekte verschicken, neue Objekte erzeugen und existierende Objekte löschen. Objekte sind grundsätzlich selbständige Ausführungseinheiten, die unabhängig voneinander und parallel arbeiten können<sup>4</sup>.

*Identität*

Objekte verhalten sich in mehreren Aspekten wie Gegenstände der materiellen Welt (bei Gegenständen denke man etwa an Autos, Lampen, Telefone, Lebewesen etc.). Insbesondere haben sie eine *Identität*: Ein Objekt kann nicht an zwei Orten gleichzeitig sein; es kann sich ändern, bleibt dabei aber dasselbe Objekt (man denke beispielsweise daran, dass ein Auto umlackiert werden kann, ohne dass sich dabei seine Identität ändert, oder dass bei einem Menschen im Laufe seines Lebens fast alle Zellen ausgetauscht werden, die Identität des Menschen davon aber unberührt bleibt). Objekte im Sinne der

<sup>4</sup>Dieses Modell wird in vielen, in der Praxis eingesetzten Programmiersprachen nicht umgesetzt. Siehe auch Absatz „Relativierung inhärenter Parallelität“ auf Seite 20

objektorientierten Programmierung unterscheiden sich also von üblichen mathematischen Objekten wie Zahlen, Funktionen, Mengen, usw. (Zahlen haben keine Lebensdauer, keinen „Aufenthaltort“ und keinen Zustand.)

Im Folgenden werden wir das oben skizzierte *objektorientierte Grundmodell* näher erläutern und den Zusammenhang zu den drei in Abschn. 1.1.2 skizzierten Anforderungen diskutieren.

*obj.-orient.  
Grundmodell*

**Modellierung der realen Welt.** Jedes der unterschiedlichen Programmierparadigmen hat sich aus einem speziellen technischen und gedanklichen Hintergrund heraus entwickelt. Die prozedurale Programmierung ist im Wesentlichen aus einer Abstraktion des Rechenmodells entstanden, das heutigen Computern zugrunde liegt. Die deklarative Programmierung basiert auf klassischen Beschreibungsmitteln der Mathematik. Ein grundlegendes Ziel der objektorientierten Programmierung ist es, eine möglichst gute softwaretechnische Modellierung der realen Welt zu unterstützen und damit insbesondere eine gute Integration von realer und softwaretechnischer Welt zu ermöglichen. Die softwaretechnisch realisierte Welt wird oft als *virtuelle* Welt bezeichnet. Das folgende Zitat (vgl. [MMPN93], Kap. 1) gibt eine Idee davon, was mit Modellierung der realen Welt gemeint ist und welche Vorteile sie mit sich bringt:

„The basic philosophy underlying object-oriented programming is to make the programs as far as possible reflect that part of the reality they are going to treat. It is then often easier to understand and to get an overview of what is described in programs. The reason is that human beings from the outset are used to and trained in the perception of what is going on in the real world. The closer it is possible to use this way of thinking in programming, the easier it is to write and understand programs.“

Anhand eines kleinen Beispiels wollen wir genauer studieren, wie eine Modellierung der realen Welt in einem Programm aussehen kann und was dafür benötigt wird:

### **Beispiel 1.2.1** (*Modellierung der Gerätesteuerung in einem Haushalt*)

*Als fiktive Aufgabe wollen wir ein System konstruieren, dass es uns gestattet, alle elektrischen und elektronischen Geräte in einem Haus zentral von einem Rechner aus zu bedienen. Den Ausschnitt aus der realen Welt, der für eine Aufgabe relevant ist, nennen wir den Aufgabenbereich. In unserem Fall besteht der Aufgabenbereich also u.a. aus Lampen, CD-Spielern, HiFi-Verstärkern, Telefonen etc. sowie aus den Räumen des Hauses. Geräte und Räume sind Objekte der realen Welt. Sie können sich in unterschiedlichen Zuständen befinden: Eine Lampe kann an- oder abgeschaltet sein, ein CD-Spieler kann leer sein oder eine CD geladen haben, der Verstärker kann in unterschiedlichen Lautstärken spielen, in einem Raum können sich verschiedene Geräte befinden. In einem objektorientierten Programm entspricht jedem (relevanten)*

*Objekt der realen Welt ein Objekt in der virtuellen Welt des Programms. Jedes Objekt hat einen Zustand.*

*In der realen Welt geschieht die Inspektion und Änderung des Zustands der Objekte auf sehr unterschiedliche Weise. Inspektion: Man schaut nach, ob eine Lampe leuchtet, hört hin, ob der CD-Spieler läuft, sucht nach dem Telefon. Änderungen: Man schaltet Geräte an und aus, stellt sie lauter, legt CD's in den CD-Spieler, stellt neue Lampen auf. In der virtuellen Welt stellen die Objekte Operationen zur Verfügung, um ihren Zustand zu inspizieren und zu verändern. Beispielsweise wird es für Lampen eine Operation geben, mit der festgestellt werden kann, ob die Lampe leuchtet, und eine Operation, um sie an- bzw. auszuschalten. □*

Durch die Aufteilung der Verarbeitungsprozesse auf mehrere Objekte ermöglicht das Grundmodell der objektorientierten Programmierung vom Konzept her insbesondere eine natürliche Behandlung von parallelen und verteilten Prozessen der realen Welt.

**Relativierung inhärenter Parallelität** Das Konzept der inhärenten Parallelität wird von den meisten objektorientierten Programmiersprachen allerdings **nicht** unterstützt. Methoden werden standardmäßig sequentiell ausgeführt, auch über Objektgrenzen hinweg. Parallelität muss in diesen Sprachen, ähnlich wie in imperativen Programmiersprachen, explizit durch Sprachkonstrukte eingeführt werden. In Java gibt es dafür das Konzept der Threads, das in Abschnitt 6.2 ausführlich vorgestellt wird.

**Programmstruktur.** Objekte bilden eine Einheit aus Daten und den auf ihnen definierten Operationen. Die Gesamtheit der Daten in einem objektorientierten Programm ist auf die einzelnen Objekte verteilt. Auch gibt es keine global arbeitenden Prozeduren bzw. Operationen<sup>5</sup>. Jede Operation gehört zu einem Objekt und lässt sich von außen nur über das Schicken einer Nachricht an dieses Objekt auslösen. Abbildung 1.3 skizziert dieses Modell in vereinfachter Form für zwei Objekte *obj1* und *obj2*. Beide Objekte haben objektlokale Variablen, sogenannte *Attribute*, z.B. hat *obj1* die Attribute *a1* und *a2*. Beide Objekte haben lokale Operationen, sogenannte *Methoden*: *obj1* hat die Methoden *m*, *m1*, *m2*; *obj2* hat die Methoden *m* und *n* (auf die Angabe der Methodenrümpfe wurde verzichtet). Die Kommunikation zwischen Objekten ist in Abb. 1.3 (wie auch im Folgenden) durch einen gepunkteten Pfeil dargestellt: Objekt *obj1* schickt *obj2* die Nachricht *m* mit den Parametern *1618* und *"SS2007"*. Objekt *obj2* führt daraufhin seine Methode *m* aus und liefert (möglicherweise) ein Ergebnis zurück.

Attribut

<sup>5</sup>Einige Programmiersprachen weichen von diesem Grundkonzept ab und bieten sogenannte *Klassenmethoden* an (siehe auch Abschnitt 2.1.4.2). Klassenmethoden ähneln Prozeduren der prozeduralen Programmierung und können unabhängig von einem Objekt aufgerufen werden.

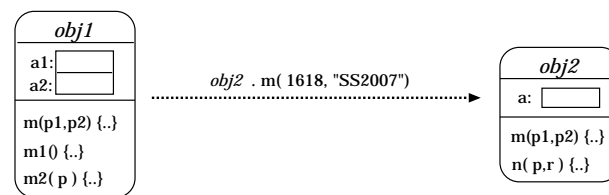


Abbildung 1.3: Kommunizierende Objekte

Das Grundmodell der objektorientierten Programmierung geht demnach davon aus, dass die zu verarbeitenden Informationen auf Objekte verteilt sind. Die Verarbeitung der Information geschieht entweder objektlokal oder durch Nachrichtenkommunikation zwischen Objekten. Dabei werden zur Beschreibung der objektlokalen Verarbeitung meist prozedurale Techniken verwendet.

Jedes Objekt hat eine klar festgelegte Schnittstelle, die seine Eigenschaften beschreibt. Diese Eigenschaften bestehen aus den Nachrichten, die es „versteht“, d.h. für die es eine passende Methode besitzt und aus seinen von außen direkt zugreifbaren Attributen. Idealerweise werden statt direkt zugreifbarer Attribute Zugriffsmethoden verwendet, über die die Attribute gelesen und geschrieben werden können. Diese inhärente Schnittstellenbildung bringt zwei zentrale Fähigkeiten der objektorientierten Programmierung mit sich: Datenkapselung und Klassifizierung. Wenn auf den Zustand eines Objekts von anderen Objekten nur über Methoden zugegriffen wird, hat ein Objekt die vollständige Kontrolle über seine Daten, sofern es nicht von außen Referenzen auf andere Objekte übergeben bekommt, die einen Teil seines Zustandes ausmachen.<sup>6</sup> Insbesondere kann es die Konsistenz zwischen Attributen gewährleisten und verbergen, welche Daten es in Attributen hält und welche Daten es erst auf Anforderung berechnet.

Objekte können nach ihrer Schnittstelle klassifiziert werden. Beispielsweise könnten die Objekte *obj1* und *obj2* aus Abb. 1.3 zu der Klasse der Objekte zusammengefasst werden, die die Nachricht *m* (mit zwei Parametern) verstehen. Typisches Beispiel wäre eine Klasse von Objekten, die alle eine Methode *drucken* besitzen, um sich selbst auszudrucken. Solche Klassifikationen kann man insbesondere nutzen, um Mengen von Objekten hierarchisch zu strukturieren. Betrachten wir als Beispiel die Personengruppen an einer Universität, skizziert in Abb. 1.4: Jedes Objekt der Klasse *Person* hat eine Methode, um Namen und Geburtsdatum zu erfragen. Jeder Student kann darüber hinaus nach Matrikelnummer und Semesterzahl befragt werden; bei den Angestellten kommen stattdessen Angaben über das Arbeitsverhältnis und die Zuordnung zu Untergliederungen der Universität hinzu.

<sup>6</sup>Der Zustand solcher Objekte kann dann über weiterhin bestehende Referenzen von außen ebenfalls verändert werden. Auf diese Problematik wird im weiteren Verlauf des Kurses noch eingegangen.



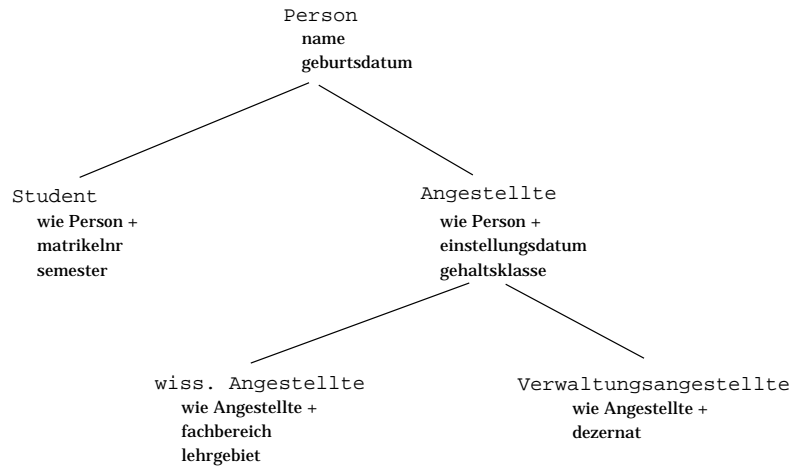


Abbildung 1.4: Klassifikation der Personen an einer Universität

Wie wir im Folgenden sehen werden, lässt sich eine Klassifikation wie in Abb. 1.4 in objektorientierten Programmen direkt modellieren. Für das obige Beispiel heißt das, dass man die Eigenschaften, die allen Personen gemeinsam sind, nur einmal zu beschreiben braucht und sie an alle untergeordneten Personenklassen „vererben“ kann.

**Erweiterung von Programmen.** Das Nachrichtenmodell in der objektorientierten Programmierung bringt wesentliche Vorteile für eine gute Erweiterbarkeit und Wiederverwendbarkeit von Programmen mit sich. Betrachten wir dazu ein Programm, das Behälter für druckbare Objekte implementiert, d.h. für Objekte, die die Nachricht `drucken` verstehen. Die Behälterobjekte sollen eine Methode `alle_drucken` besitzen, die allen Objekten im Behälter die Nachricht `drucken` schickt. Der Programmtext der Methode `alle_drucken` braucht nicht geändert zu werden, wenn das Programm neue druckbare Objekte mit anderen Eigenschaften unterstützen soll, da jedes druckbare Objekt seine eigene, spezifische Methode zum Drucken besitzt und damit auf die Nachricht `drucken` vom Behälterobjekt reagieren kann.

Um zu sehen, dass derartige Erweiterungseigenschaften nicht selbstverständlich sind, betrachten wir das gleiche Problem im Rahmen der prozeduralen Programmierung. Dazu gehen wir von einem Pascal-Programmfragment aus, das die Daten von Studenten und Angestellten druckt, d.h. die Personen fungieren hier als druckbare Objekte. Als Behälter wählen wir ein Feld mit Elementen vom Typ `Druckbar`. Da die spezifischen Druckprozeduren der Personenarten in Pascal nicht den Personen zugeordnet werden können, muss explizit eine Fallunterscheidung programmiert werden, die für jede Personenart einen Fall mit dem entsprechenden Prozeduraufruf vorsieht:

```
const anz = 100;
```

```

type Student    = record    ... end;
  WissAng       = record    ... end;
  VerwAng       = record    ... end;

ObjektArt = ( stud, wiss, verw );
Druckbar  = record case art: ObjektArt of
    stud: ( s: Student );
    wiss: ( w: WissAng );
    verw: ( v: VerwAng );
end;
Behaelter = Array [1 .. anz] of Druckbar;

procedure Student_drucken( s: Student );
begin ... end;

procedure WissAng_drucken( w: WissAng );
begin ... end;

procedure VerwAng_drucken( v: VerwAng );
begin ... end;

procedure alle_drucken( b: Behaelter );
  var e      : Druckbar;
      index  : Integer;
begin
  ...
  for index := 1 to anz do
  begin
    e := b[index];
    case e.art of
      stud: Student_drucken( e.s );
      wiss: WissAng_drucken( e.w );
      verw: VerwAng_drucken( e.v );
    end
  end { for-Schleife }
end;

```

Die Entscheidung darüber, welche spezifische Druckprozedur auszuführen ist, wird in der Prozedur `alle_drucken` getroffen. Damit zieht jede Erweiterung des Typs `Druckbar` um neue Objektarten eine Veränderung der Fallunterscheidung in der Prozedur `alle_drucken` nach sich. (Insbesondere muss die Prozedur `alle_drucken` jedes Mal neu übersetzt werden.) In der objektorientierten Programmierung wird diese Fallunterscheidung implizit vom Nachrichtenmechanismus übernommen. Durch den Nachrichtenmechanismus wird die Bindung zwischen der Anforderung eines Dienstes (Verschicken einer Nachricht) und dem ausführenden Programmteil (Methode)

*dynamisches /  
statisches Bin-  
den*

erst zur Programmlaufzeit getroffen (dynamisch), beim Prozeduraufruf zur Übersetzungszeit (statisch).

*Subtyping*

Ein weiterer zentraler Aspekt des objektorientierten Grundmodells für die Erweiterung von Programmen resultiert aus der Bündelung von Daten und Operationen zu Objekten mit klar definierten Schnittstellen (siehe Paragraph „Programmstruktur“ auf Seite 20). Ein Objekt beziehungsweise seine Beschreibung lässt sich einfach erweitern und insbesondere leicht spezialisieren. Dazu fügt man zusätzliche Attribute und/oder Methoden hinzu bzw. passt existierende Methoden an neue Erfordernisse an. Insgesamt erhält man Objekte mit einer umfangreicheren Schnittstelle, sodass die erweiterten Objekte auch an allen Stellen eingesetzt werden können, an denen die alten Objekte zulässig waren. Auf diese Möglichkeit wird im Zusammenhang mit *Subtyping* genauer eingegangen (s. Kapitel 3). Der Nachrichtenmechanismus mit seiner dynamischen Bindung garantiert dabei, dass die angepassten Methoden der neuen Objekte ausgeführt werden. Wie wir in späteren Kapiteln genauer untersuchen werden, ermöglicht diese Art der Programmerweiterung eine elegante Spezialisierung existierender Objekte unter Wiederverwendung des Programmcodes der alten Objekte. Diese Art der Wiederverwendung nennt man *Vererbung*: Die spezialisierten Objekte erben den Programmcode der existierenden Objekte.

## 1.3 Programmiersprachlicher Hintergrund

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

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

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

### 1.3.1 Grundlegende Sprachmittel am Beispiel von Java

Bei den meisten objektorientierten Programmiersprachen werden objektlokale Berechnungen mit imperativen Sprachmitteln beschrieben. Im Folgenden sollen die in den späteren Kapiteln benötigten Sprachmittel systematisch zusammengefasst werden, um eine begriffliche und programmtechnische Grundlage zu schaffen. Begleitend werden wir zeigen, wie diese Sprachmittel in Java umgesetzt sind. Dabei werden wir uns auf eine knapp gehaltene

Einführung beschränken, die aber alle wesentlichen Aspekte anspricht. Eine detaillierte Darstellung findet sich in [GJS96].

Der Abschnitt besteht aus drei Teilen. Er erläutert zunächst den Unterschied zwischen Objekten und Werten. Dann geht er näher auf Werte, Typen und Variablen ein; in diesem Zusammenhang beschreibt er die Basisdatentypen von Java und definiert, was Ausdrücke sind und wie sie in Java ausgewertet werden. Der letzte Teil behandelt Anweisungen und ihre Ausführung. Ziel dieses Abschnitts ist es u.a., den Leser in die Lage zu versetzen, selbständig imperative Programme in Java zu schreiben.

### 1.3.1.1 Objekte und Werte: Eine begriffliche Abgrenzung

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

*Objekt*  
*vs. Wert*

1. Zustand: Objekte haben einen veränderbaren Zustand (ein Auto kann eine Beule bekommen; ein Mensch eine neue Frisur; eine Mülltonne kann geleert werden). Werte sind abstrakt und können nicht verändert werden (es macht keinen Sinn, die Zahl 37 verändern zu wollen; entnehme ich einer Menge ein Element, erhalte ich eine andere Menge).
2. Identität: Objekte besitzen eine Identität, die vom Zustand unabhängig ist. Objekte können sich also völlig gleichen, ohne identisch zu sein (man denke etwa an zwei baugleiche Autos). Insbesondere kann man durch Klonen/Kopieren eines Objekts *obj* ein anderes Objekt erzeugen, das *obj* in allen Eigenschaften gleicht, aber nicht mit ihm identisch ist. Zukünftige Änderungen des Zustands von *obj* haben dann keinen Einfluss auf den Zustand der Kopie.
3. Lebensdauer: Objekte besitzen eine Lebensdauer; insbesondere gibt es Operationen, um Objekte zu erzeugen, ggf. auch um sie zu löschen. Werte besitzen keine beschränkte Lebensdauer, sondern existieren quasi ewig.
4. Aufenthaltsort: Objekten kann man üblicherweise einen Aufenthaltsort, beschrieben durch eine Adresse, zuordnen. Werte lassen sich nicht lokalisieren.
5. Verhalten: Objekte reagieren auf Nachrichten und weisen dabei ein zustandsabhängiges Verhalten auf. Werte besitzen kein „Eigenleben“; auf

ihnen operieren Funktionen, die Eingabewerte zu Ergebniswerten in Beziehung setzen.

Der konzeptionell relativ klare Unterschied zwischen Objekten und Werten wird bei vielen programmiersprachlichen Realisierungen nur zum Teil beibehalten. Zur Vereinheitlichung behandelt man Werte häufig wie Objekte. So werden z.B. in Smalltalk ganze Zahlen als Objekte modelliert. Zahl-Objekte besitzen einen unveränderlichen Zustand, der dem Wert der Zahl entspricht, eine Lebensdauer bis zum Ende der Programmlaufzeit und Methoden, die den arithmetischen Operationen entsprechen. Zahl-Objekte werden als identisch betrachtet, wenn sie denselben Wert repräsentieren. In Java wird eine ähnliche Konstruktion für Zeichenreihen verwendet und um zahlwertige Datentypen als Subtypen des ausgezeichneten Typs `Object` behandeln zu können (vgl. Unterabschn. 3.2.2.1). Andererseits lassen sich Objekte auch durch Werte formal modellieren, beispielsweise als ein Paar bestehend aus einem Objektbezeichner (Bezeichner sind Werte) und einem Wert, der den Zustand repräsentiert.

### 1.3.1.2 Objektreferenzen, Werte, Felder, Typen und Variablen

Dieser Unterabschnitt behandelt den Unterschied zwischen Objekten und Objektreferenzen, beschreibt die Werte und Typen von Java, deren Operationen sowie die Bildung und Auswertung von Ausdrücken.

**Objekte und Objektreferenzen.** Die Beschreibung und programmtechnische Deklaration von Objekten wird ausführlich in Kap. 2 behandelt. Um präzise erklären zu können, was Variablen in Java enthalten können, müssen wir allerdings schon hier den Unterschied zwischen Objekten und Objektreferenzen erläutern. Abbildung 1.5 zeigt das Objekt *obj* und die Variablen *a*, *b*, *i* und *flag*. Wie in der Abbildung zu sehen, stellen wir Objekte als Rechtecke mit runden Ecken und Variablen als Rechtecke mit rechtwinkligen Ecken dar. Die Variablen *a* und *b* enthalten eine *Referenz* auf *obj*, die graphisch jeweils durch einen Pfeil gezeichnet sind. Während wir Objektreferenzen durch Pfeile repräsentieren, benutzen wir bei anderen Werten die übliche Darstellung. So enthält die `int`-Variable *i* den Wert 1998 und die boolesche Variable *flag* den Wert `true`.

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

**Werte, Typen und Variablen in Java.** Ein Datentyp beschreibt eine Menge von Werten zusammen mit den darauf definierten Operationen. Java stellt die

Objekt-  
referenz

Zeiger

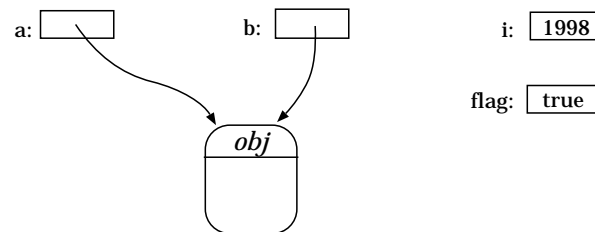


Abbildung 1.5: Referenziertes Objekt und Variablen

vordefinierten Basisdatentypen `byte`, `short`, `int`, `long`, `float`, `double`, `char` und `boolean` zur Verfügung. Wertebereich und der jeweilige Speicherbedarf der Basisdatentypen sind in Abb. 1.6 zusammengestellt.

*Basis-  
datentypen*

Typname	Wertebereich und Notation	Speicherbedarf
<code>byte</code>	-128 bis 127	1 Byte
<code>short</code>	-32768 bis 32767	2 Byte
<code>int</code>	-2147483648 bis 2147483647	4 Byte
<code>long</code>	-9223372036854775808L bis 9223372036854775807L	8 Byte
<code>float</code>	im Bereich $\pm 3.402823\text{E}+38\text{F}$ jeweils 6-7 signifikante Stellen	4 Byte
<code>double</code>	im Bereich $\pm 1.797693\text{E}+308$ jeweils 15 signifikante Stellen	8 Byte
<code>char</code>	65.536 Unicode-Zeichen, Notationsbeispiele: 'a', '+', '\n', '\'', '\u0022'	2 Byte
<code>boolean</code>	true, false	1 Byte

Abbildung 1.6: Basisdatentypen von Java

Abbildung 1.6 zeigt auch, wie die *Konstanten* der Basisdatentypen in Java geschrieben werden. Die Konstanten der Typen `byte`, `short` und `int` werden notationell nicht unterschieden. Die Konstanten des Typs `long` haben ein „L“ als Postfix. Bei Gleitkommazahlen kann die Exponentenangabe entfallen. Unicode-Zeichen lassen sich grundsätzlich durch `'\uxxxx'` in Java ausdrücken, wobei *x* für eine Hexadezimalziffer steht; jedes ASCII-Zeichen *a* lässt sich aber auch direkt als `'a'` notieren; darüber hinaus bezeichnet `'\t'` das Tabulatorzeichen, `'\n'` das Neue-Zeile-Zeichen, `'\''` das Apostroph, `'\"'` das Anführungszeichen und `'\\'` den Backslash.)

*Konstante*

Ein *Wert* in Java ist entweder

*Werte  
in Java*

- ein Element eines der Basisdatentypen,
- eine Objektreferenz oder

- die spezielle Referenz `null`, die auf kein Objekt verweist.

Typen  
in Java

Jeder Wert in Java hat einen *Typ*. Beispielsweise hat der durch die Konstante `true` bezeichnete Wert den Typ `boolean`; die Konstanten `'c'` und `'\n'` bezeichnen Werte vom Typ `char`. Der Typ charakterisiert die Operationen, die auf den Werten des Typs zulässig sind. Außer den vordefinierten Basisdatentypen gibt es in Java Typen für Objekte. Die Typisierung von Objekten behandeln wir in den Kapiteln 2 und 3. Für die hier zusammengestellten Grundlagen ist es nur wichtig, dass jedes Objekt in Java einen Typ hat und dass nicht zwischen dem Typ eines Objekts *obj* und dem Typ der Referenz auf *obj* unterschieden wird<sup>7</sup>. In Java ist vom Programmkontext her immer klar, wann ein Objekt und wann eine Objektreferenz gemeint ist. Deshalb werden auch wir, wie in vielen anderen Texten über Java üblich, ab Kap. 3 nicht mehr zwischen Objekten und Objektreferenzen unterscheiden, um die Sprechweise zu vereinfachen. In diesem Kapitel werden wir weiterhin präzise formulieren; in Kap. 2 werden wir auf den Unterschied durch einen Klammerzusatz aufmerksam machen, wo dies ohne Umstände möglich ist.

Variablen

*Variablen* sind Speicher für Werte. In Java sind Variablen typisiert. Variablen können nur Werte speichern, die zu ihrem Typ gehören. Eine *Variablen-deklaration* legt den Typ und Namen der Variablen fest. Folgendes Programmfragment deklariert die Variablen `i`, `flag`, `a`, `b`, `s1` und `s2`:

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

Die Variable `i` kann Zahlen vom Typ `int` speichern; `flag` kann boolesche Werte speichern; `a` und `b` können Referenzen auf Objekte vom Typ `Object`, und `s1` und `s2` können Referenzen auf Objekte vom Typ `String` speichern. Wie wir in Kap. 2 sehen werden, sind `Object` und `String` vordefinierte Objekttypen.

Komponenten-  
typ  
Typkonstruktor

**Felder.** *Felder* (engl. *arrays*<sup>8</sup>) sind in Java Objekte; dementsprechend werden sie dynamisch, d.h. zur Laufzeit, erzeugt und ihre Referenzen können an Variablen zugewiesen und als Parameter an Methoden übergeben werden. Ist *T* ein beliebiger Typ in Java, dann bezeichnet `T[]` den Typ der Felder mit *Komponententyp* *T*. Den Operator `[]` nennt man einen *Typkonstruktor*, da er aus einem beliebigen Typ einen neuen Typ konstruiert.

<sup>7</sup>In C++ werden Objekte und Objektreferenzen typmäßig unterschieden.

<sup>8</sup>Zur Beachtung: „Arrays“ sind nicht mit „fields“ zu verwechseln. In der engl. Java-Literatur wird das Wort „field“ für die in einer Klasse deklarierten Attribute verwendet.

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

*Feldelement*

Felder als Objekte zu realisieren hat den Vorteil, dass sich Felder auf diese Weise besser in objektorientierte Klassenhierarchien einbetten lassen (Genaueres dazu in Kap. 3). Außerdem gewinnt man an Flexibilität, wenn man zwei- bzw. mehrdimensionale Felder realisieren möchte. In zweidimensionalen Feldern, wie man sie beispielsweise in Pascal deklarieren kann, müssen alle Spalten bzw. alle Zeilen die gleiche Länge haben. Dies gilt nicht bei der Java-Realisierung mittels Referenzen. Wenn die von einem Feld referenzierten Felder nicht alle gleich lang sind, spricht man auch von *Ragged Arrays*.

Auch kann man mehrfach auftretende „Spalten“ bzw. „Zeilen“ durch ein mehrfach referenziertes Feld realisieren (vgl. Abbildung 1.8). Der Preis für die größere Flexibilität ist im Allgemeinen größerer Speicherbedarf, höhere Zugriffszeiten sowie eine etwas gestiegene Programmierkomplexität und damit Fehleranfälligkeit.

Folgendes Programmfragment deklariert Variablen für zwei eindimensionale Felder `vor` und `Fliegen` und ein zweidimensionales Feld `satz`:

```
char[] vor;
char[] Fliegen;
char[][] satz;
```

(Um syntaktische Kompatibilität zur Sprache C zu erreichen, darf der Typkonstruktor `[]` auch erst nach dem deklarierten Namen geschrieben werden, also `char vor[]` statt `char[] vor`.) Obige Variablendeklarationen legen lediglich fest, dass die Variablen `vor` und `Fliegen` jeweils eine Referenz auf ein Feld (beliebiger Länge) enthalten können, deren Elemente vom Typ `char` sind und dass `satz` eine Referenz auf ein Feld mit Komponententyp `char[]` enthalten kann; d.h. die Feldelemente können (Referenzen auf) `char`-Felder speichern. Die Anzahl der Feldelemente wird in diesen Deklarationen noch nicht festgelegt. Wie eine solche Festlegung erfolgt, wird im Rahmen von Zuweisungen (Abbildung 1.7) erläutert.

**Operationen, Zuweisungen und Auswertung in Java.** Java stellt drei Arten von Operationen zur Verfügung:

1. Operationen, die auf allen Typen definiert sind. Dazu gehören die Gleichheitsoperation `==` und die Ungleichheitsoperation `!=` mit booleischem Ergebnis sowie die *Zuweisungsoperation* `=`.

*Zuweisung*



2. Operationen, die nur für Objektreferenzen bzw. Objekte definiert sind (Methodenaufruf, Objekterzeugung, Typtest von Objekten); diese werden wir in Kap. 2 behandeln.
3. Operationen zum Rechnen mit den Werten der Basisdatentypen; diese sind in Abb. 1.9 zusammengefasst.

**Zuweisungen** In Java (wie in C) ist die Zuweisung eine Operation mit Seiteneffekt: Sie weist der Variablen auf der linken Seite des Zuweisungszeichens „`=`“ den Wert des Ausdrucks der rechten Seite zu und liefert diesen Wert als Ergebnis der gesamten Zuweisung zurück. Dazu betrachten wir diverse Beispiele aufbauend auf den Variablendeklarationen von S. 28 und 29; der Ergebniswert ist teilweise hinter dem *Kommentarzeichen* `//` angegeben, das den Rest der Zeile als Kommentar kennzeichnet:

*Kommentar-  
zeichen*

```
(1) i = 4;                // nach Auswertung: i==4   Ergebnis: 4
(2) flag = (5 != 3);     // nach Ausw.: flag==true  Ergebnis: true
(3) flag = (5 != (i=3)); // nach Ausw.: i==3,   flag==true
                        // Ergebnis: true
(4) a = (b = null);      /* nach Ausw.: a==null, b==null
                        Ergebnis: null */
(5) vor = new char[3];
(6) vor[0] = 'v';
(7) vor[1] = 'o';
(8) vor[2] = 'r';
(9) char[] Fliegen = {'F','l','i','e','g','e','n'};
(10) char[][] satz = { Fliegen,
                      {'f','l','i','e','g','e','n'},
                      vor,
                      Fliegen };
```

Abbildung 1.7: Zuweisungen

*Kommentar-  
klammern*

Die mit (4) markierte Zeile demonstriert außerdem, dass in Java auch die *Kommentarklammern* `/*` und `*/` benutzt werden können.

Die Zeilen (5) - (10) zeigen alle im Zusammenhang mit Feldern relevanten Operationen. In Zeile (5) wird die Variable `vor` mit einem Feld der Länge 3 initialisiert. In den Zeilen (6)-(8) werden die drei Feldelemente initialisiert; man beachte, dass die Feldelemente von 0 bis *length* - 1 indiziert werden. In Zeile (9) wird die Variable `Fliegen` deklariert und mit einem Feld der Länge 7 initialisiert; die Länge berechnet der Übersetzer dabei aus der Länge der angegebenen Liste von Ausdrücken (in dem Beispiel ist jeder Ausdruck eine `char`-Konstante). Das Initialisieren von Arrays in dieser Form mit geschweiften Klammern ohne explizite Angabe der Arraylänge ist nur bei gleichzeitiger



Deklaration der Variable gestattet. Die schon auf S. 29 gesehene Deklaration darf erst hier erstmalig auftreten. In Zeile (10) wird die Variable `satz` deklariert und mit einem vierelementigen Feld initialisiert,

- deren erstes und viertes Element mit dem in Zeile (9) erzeugten Feld initialisiert wird,
- deren zweites Element mit einem neu erzeugten Feld initialisiert wird
- und deren drittes Element mit dem von `vor` referenzierten Feld initialisiert wird.

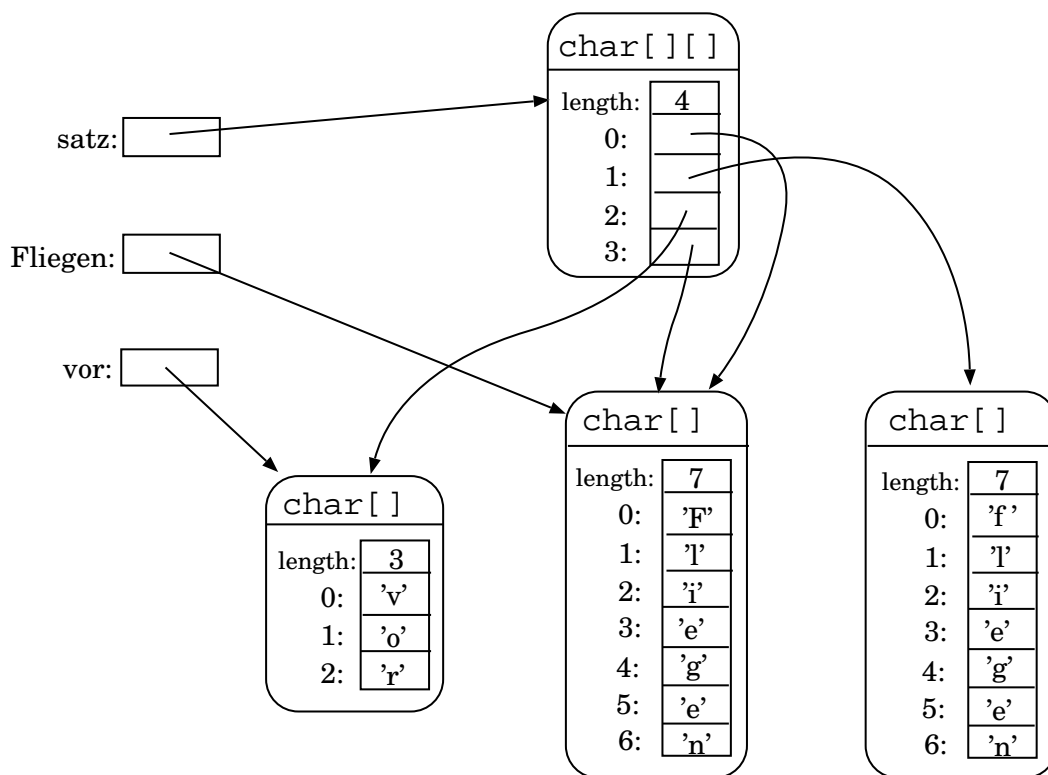


Abbildung 1.8: Die Feldobjekte zum Beispiel

Das resultierende *Objektgeflecht* ist in Abb. 1.8 dargestellt.

**Operationen auf Basisdatentypen** Die Operationen auf Basisdatentypen sind in der folgenden Tabelle zusammengefasst.

Operator	Argumenttypen	Ergebnistyp	Beschreibung
$+, -, *, /, \%$	$\text{int} \times \text{int}$	$\text{int}$	ganzzahlige Addition, etc.
$+, -, *, /, \%$	$\text{long} \times \text{long}$	$\text{long}$	ganzzahlige Addition, etc., wobei $\%$ den Rest bei ganzzahliger Division liefert
$+, -, *, /$	$\text{float} \times \text{float}$	$\text{float}$	Gleitkomma-Addition, etc.
$+, -, *, /$	$\text{double} \times \text{double}$	$\text{double}$	Gleitkomma-Addition, etc.
$-$	<i>zahltyp</i>	<i>zahltyp</i>	arithmetische Negation
$<, <=, >, >=$	$\text{zahltyp} \times \text{zahltyp}$	$\text{boolean}$	kleiner, kleiner-gleich, etc., wobei <i>zahltyp</i> für $\text{int}$ , $\text{long}$ , $\text{float}$ oder $\text{double}$ steht
$!$	$\text{boolean}$	$\text{boolean}$	logisches Komplement
$\&,  , ^$	$\text{boolean} \times \text{boolean}$	$\text{boolean}$	logische Operationen Und, Oder und ausschließendes Oder (xor)
$\&\&,   $	$\text{boolean} \times \text{boolean}$	$\text{boolean}$	nicht-strikte Und-/Oder- Operation, dh. rechter Operand wird ggf. nicht ausgewertet
$\_?\_:\_$	$\text{boolean} \times \text{typ} \times \text{typ}$	<i>typ</i>	bedingter Ausdruck (Bedeutung auf Seite 33 erläutert)

Abbildung 1.9: Operationen der Basisdatentypen

Die obige Abbildung stellt nicht alle Operationen in Java dar: Der  $++$ -Operator im Zusammenhang mit Objekten des Typs `String` wird in Kap. 2 erläutert; Operationen, die auf der Bitdarstellung von Zahlen arbeiten, sowie bestimmte Formen der Zuweisung und Operatoren zum Inkrementieren und Dekrementieren haben wir der Kürze halber weggelassen.

#### Ausdrücke

**Ausdrücke und deren Auswertung** Ein *Ausdruck* (engl. *expression*) ist eine Variable, eine Konstante oder eine Operation angewendet auf Ausdrücke. Wie üblich werden Klammern verwendet, um die Reihenfolge der anzuwendenden Operationen eindeutig zum Ausdruck zu bringen. Außerdem werden, um Klammern einzusparen, gewisse Vorrangregeln beachtet. Z. B. hat  $*$  Vorrang vor  $+$  („Punktrechnung vor Strichrechnung“) und alle arithmetischen und logischen Operatoren haben Vorrang vor dem Zuweisungsoperator  $=$ . Jeder Ausdruck in Java besitzt einen Typ, der sich bei Variablen aus deren Deklaration ablesen lässt, der sich bei Konstanten aus der Notation ergibt und der bei Operationen dem Ergebnistyp entspricht. Grundsätzlich gilt, dass die Typen der Operanden genau den Argumenttypen der Operationen entsprechen müssen. Um derartige Typgleichheit erzielen zu können, bietet Java die Möglichkeit, Werte eines Typs in Werte eines anderen Typs

zu konvertieren. Häufig spricht man anstatt von *Typkonvertierung* auch von Typecasts oder einfach nur von Casts (vom engl. to cast).

*Typkonvertierung  
(cast)*

In Java werden Typkonvertierungen dadurch notiert, dass man den Namen des gewünschten Ergebnistyps, in Klammern gesetzt, dem Wert bzw. Ausdruck voran stellt. Beispielsweise bezeichnet `(long) 3` den Wert drei vom Typ `long`, ist also gleichbedeutend mit `3L`. Java unterstützt insbesondere die Typkonvertierung zwischen allen Zahltypen, wobei der Typ `char` als Zahltyp mit Wertebereich 0 bis 65.536 betrachtet wird. Vergrößert sich bei der Typkonvertierung der Wertebereich, z.B. von `short` nach `long`, bleibt der Zahlwert unverändert. Andernfalls, z.B. von `int` nach `byte`, führt die Typkonvertierung im Allgemeinen zu einer Verstümmelung des Wertes. Um die Lesbarkeit der Programme zu erhöhen, werden bestimmte Konvertierungen in Java implizit vollzogen: Z.B. werden alle ganzzahligen Typen, wo nötig, in ganzzahlige Typen mit größerem Wertebereich konvertiert und ganzzahlige Typen werden, wo nötig, in Gleitkommatypen konvertiert. Die folgenden beiden Beispiele demonstrieren diese implizite Typkonvertierung, links jeweils der Java-Ausdruck, bei dem implizit konvertiert wird, rechts ist die Konvertierung explizit angegeben:

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

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

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

*Auswertung  
von  
Ausdrücken*

```
B ? A1 : A2
```

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

Die Auswertung eines Ausdrucks kann in Java auf zwei Arten terminieren: *normal* mit dem üblichen Ergebnis oder *abrupt*, wenn bei der Ausführung einer Operation innerhalb des Ausdrucks ein Fehler auftritt – z.B. Division durch null. Im Fehlerfall wird die Ausführung des gesamten umfassenden

*normale und  
abrupte  
Terminierung  
der  
Auswertung*

Ausdrucks sofort beendet und eine Referenz auf ein Objekt zurückgeliefert, das Informationen über den Fehler bereitstellt (vgl. den folgenden Abschnitt über Kontrollstrukturen sowie die Kapitel 2 und 4).

### 1.3.1.3 Anweisungen, Blöcke und deren Ausführung

Anweisungen dienen dazu, den Kontrollfluss von Programmen zu definieren. Während Ausdrücke *ausgewertet* werden – üblicherweise mit Rückgabe eines Ergebniswertes –, spricht man bei Anweisungen von *Ausführung* (engl. *execution*). Dieser Unterabschnitt stellt die wichtigsten Anweisungen von Java vor und zeigt, wie aus Deklarationen und Anweisungen Blöcke gebildet werden können. Wir unterscheiden *elementare* und *zusammengesetzte* Anweisungen.

Die zentrale elementare Anweisung in Java ist ein Ausdruck. Wie oben erläutert, können durch Ausdrücke in Java sowohl Zuweisungen, als auch Methodenaufrufe beschrieben werden. Insofern stellen Java-Ausdrücke<sup>9</sup> eine syntaktische Verallgemeinerung der üblichen elementaren Anweisungsformen „Zuweisung“ und „Prozeduraufruf“ dar. Ausdrücke, die als Anweisung fungieren, werden mit einem Semikolon abgeschlossen; dazu drei Beispiele:

*Ausdrücke  
als Anwei-  
sung*

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

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

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

*Blöcke*

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

<sup>9</sup>Gleiches gilt für Ausdrücke in C oder C++.

Variablendeklarationen lassen sich mit einer nachfolgenden Zuweisung zusammenfassen. Beispielsweise könnte man die Deklaration von `flag` mit der Zuweisung wie folgt verschmelzen:

```
boolean flag = ( i >= 2 );
```

Variablendeklarationen in Blöcken sind ab der Deklarationsstelle bis zum Ende des Blockes gültig.

**Klassische Kontrollstrukturen.** *Bedingte Anweisungen* gibt es in Java in den Formen:

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

wobei die Zweige der `if`-Anweisung selbstverständlich wieder zusammengesetzte Anweisungen sein können. Bei der ersten Form wird *Anweisung* nur ausgeführt, wenn der boolesche Ausdruck zu `true` ausgewertet. Im zweiten Fall wird *Anweisung1* ausgeführt, wenn der boolesche Ausdruck zu `true` ausgewertet, andernfalls wird *Anweisung2* ausgeführt. Im folgenden Beispiel wird die erste Form der `if`-Anweisung in Zeile 2 verwendet und die zweite Form in Zeile 4.

*bedingte  
Anweisung*

```
(1)  int  n = 4;
(2)  if ((n % 2) == 0)
      System.out.println ("n ist durch 2 teilbar");
(3)  n = n + 2;
(4)  if ((n % 3) == 0)
      System.out.println ("n ist durch 3 teilbar");
      else
      System.out.println ("n ist NICHT durch 3 teilbar");
```

Dabei gibt `System.out.println(...)` den als Parameter übergebenen Wert auf der Standardausgabe an der aktuellen Position aus und macht anschließend noch einen Zeilenvorschub.

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

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

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

*Schleifen-  
Anweisung*

Die `do`-Anweisung unterscheidet sich nur dadurch, dass der Schleifenrumpf auf jeden Fall einmal vor der ersten Auswertung des booleschen Ausdrucks ausgeführt wird. Die `do`-Anweisung wird beendet, sobald der boolesche Ausdruck zu `false` auswertet.

Die Ausführung der `for`-Schleife in der ersten Form kann mit Hilfe der `while`-Schleife erklärt werden; dazu betrachten wir ein kleines Programmfragment, mit dem die Fakultät berechnet werden kann (bei Eingaben größer als 20 tritt allerdings ein Überlauf auf, sodass das Programm nur im Bereich 0 bis 20 korrekt arbeitet):

```
(1)  int  n;          // Eingabeparameter
(2)  long result;    // Ergebnis
(3)  int  i;          // Schleifenvariable

(4)  n = 19;         // initialisieren mit Wert von 0 bis 20
(5)  result = 1;
(6)  for( i = 2; i <= n; i = i + 1 ) result = result * i;
(7)  // result enthaelt  fac(n)
```

Äquivalent zu der `for`-Schleife in Zeile (6) ist folgendes Fragment:

```
i = 2;
while( i <= n ) {
    result = result * i;
    i = i + 1;
}
```

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

```
String[] satz = { "Wenn", "Fliegen", "hinter", "Fliegen",
                  "fliegen", ",", "fliegen", "Fliegen",
                  "Fliegen", "nach", "." };
System.out.println();

// Lesen Sie: Fuer jedes (for each) 'wort' in 'satz'
for( String wort : satz ) {
    System.out.print(wort);
    System.out.print(" ");
}
System.out.println();
```

Ähnlich `System.out.println(...)` gibt `System.out.print(...)` den als Parameter übergebenen Wert auf der Standardausgabe an der aktuellen Position aus. Allerdings wird kein Zeilenvorschub vorgenommen.

Dasselbe Verhalten kann wie folgt mit der ersten Form der `for`-Schleife erreicht werden:

```
String[] satz = { "Wenn", "Fliegen", "hinter", "Fliegen",  
                 "fliegen", ",", "fliegen", "Fliegen",  
                 "Fliegen", "nach", "." };  
  
System.out.println();  
for( int i = 0; i < satz.length; i = i + 1 ) {  
    System.out.print(satz[i]);  
    System.out.print(" ");  
}  
System.out.println();
```

Für die effiziente Implementierung von Fallunterscheidungen bietet Java die `switch`-Anweisung. Diese soll hier nur exemplarisch erläutert werden. Nehmen wir an, dass wir den Wert einer `int`-Variablen `n`, von der wir annehmen, dass sie nur Werte zwischen 0 und 11 annimmt, als Zahlwort ausgeben wollen, also für 0 das Wort „null“ usw. Ein typischer Fall für die `switch`-Anweisung:

*switch-  
Anweisung*

```
switch ( n ) {  
case 0:  System.out.print("null");  
        break;  
case 1:  System.out.print("eins");  
        break;  
...  
case 11: System.out.print("elf");  
        break;  
default: System.out.print("n nicht zwischen 0 und 11");  
}
```

Zur Ausführung der `switch`-Anweisung wird zunächst der ganzzahlige Ausdruck hinter dem Schlüsselwort `switch` ausgewertet, in obigem Fall wird also der Wert von `n` genommen. Sofern kein Fall für diesen Wert angegeben ist, wird die Ausführung nach dem Schlüsselwort `default` fortgesetzt. Andernfalls wird die Ausführung bei dem entsprechenden Fall fortgesetzt. Die `break`-Anweisungen im obigen Beispiel beenden die Ausführung der einzelnen Fälle und brechen die `switch`-Anweisung ab; die Ausführung wird dann mit der Anweisung fortgesetzt, die der `switch`-Anweisung folgt. Fehlten die `break`-Anweisungen im Beispiel, würden alle Zahlworte mit Werten größer gleich `n` sowie die Meldung des `default`-Falls ausgegeben werden. Im Allgemeinen dient die `break`-Anweisung dazu, die umfassende Anweisung direkt

*break-  
Anweisung*



zu verlassen; insbesondere kann sie auch zum Herausspringen aus Schleifen benutzt werden.

*return-  
Anweisung*

Entsprechend kann man mittels der *return*-Anweisung die Ausführung eines Methodenrumpfs beenden. Syntaktisch tritt die *return*-Anweisung in zwei Varianten auf, je nachdem ob die zugehörige Methode ein Ergebnis zurückliefert oder nicht:

```
return ;    // in Methoden ohne Ergebnis und in Konstruktoren
return Ausdruck ;    // Wert des Ausdrucks liefert das Ergebnis
```

*normale und  
abrupte Ter-  
minierung der  
Ausführung*

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

*try-  
Anweisung*

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

```
try
    try-Block
catch ( AusnahmeTyp  Bezeichner )    catch-Block1
    ...
catch ( AusnahmeTyp  Bezeichner )    catch-BlockN
finally    finally-Block
```

Die *finally*-Klausel ist optional; die *try*-Anweisung muss allerdings immer entweder mindestens eine *catch*-Klausel oder die *finally*-Klausel enthalten. Bevor wir die Bedeutung der *try*-Anweisung genauer erläutern, betrachten wir ein kleines Beispiel. In der in Java vordefinierten Klasse `Integer` gibt es eine Methode `parseInt`, die eine Zeichenreihe als Parameter bekommt (genauer: die Methode bekommt eine Referenz auf ein `String`-Objekt als Parameter). Stellt die Zeichenreihe eine `int`-Konstante dar, terminiert die Methode normal und liefert den entsprechenden `int`-Wert als Ergebnis. Andernfalls terminiert sie abrupt und liefert ein Ausnahmeobjekt vom Typ `NumberFormatException`.

Abbildung 1.10 zeigt an einem einfachen Beispiel, wie eine solche Ausnahme behandelt werden kann.

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

Abbildung 1.10: Programmfragment zur Behandlung einer Ausnahme

Da `parseInt` den Postfix „L“ beim eingegebenen Parameter nicht akzeptiert, wird die Ausführung des Methodenaufrufs innerhalb des `try`-Blocks abrupt terminieren und eine Ausnahme vom Typ `NumberFormatException` erzeugen. Dies führt zur Ausführung der angegebenen `catch`-Klausel. Danach terminiert die gesamte `try`-Anweisung normal, sodass die Ausführung mit dem Aufruf von `println` in der letzten Zeile fortgesetzt wird.

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

```
String[] args = ...;

int m, n, ergebnis = 0;
try{
    m = Integer.parseInt( args[0] );
    n = Integer.parseInt( args[1] );
    ergebnis = m/n ;
} catch ( IndexOutOfBoundsException e ) {
    System.out.println("argf falsch initialisiert");
} catch ( NumberFormatException e ) {
    System.out.println(
        "Element in argf keine int-Konstante");
} catch ( ArithmeticException e ) {
    System.out.println("Divison durch null");
}
System.out.println( ergebnis );
...
```

Eine Ausnahme vom Typ `IndexOutOfBoundsException` tritt bei Feldzugriffen mit zu großem oder zu kleinem Index auf. Genau wie die `NumberFormatException`-Ausnahme kann solch ein falscher Feldzugriff innerhalb des `try`-Blocks zweimal vorkommen. Die Ausnahme vom Typ `ArithmeticException` fängt die mögliche Division durch null in der letzten Zeile des `try`-Blocks ab.

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

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

Dabei *passt* eine `catch`-Klausel zu einer geworfenen Ausnahme, wenn die Klasse, zu der das erzeugte Ausnahmeobjekt gehört, identisch mit oder eine untergeordnete Klasse der Klasse ist, die zur Deklaration der zu fangenden Ausnahmen innerhalb der `catch`-Klausel verwendet wurde<sup>10</sup>.

Die in den Beispielen vorkommenden Klassen `ArithmeticException`, `NumberFormatException` und `IndexOutOfBoundsException` sind alle untergeordnete Klassen von `RuntimeException`, die selbst wiederum der Klasse `Exception` untergeordnet ist. Die in Java vordefinierte Klassenhierarchie für Ausnahmetypen wird in Abschnitt 4.2.1 genauer eingeführt.

<sup>10</sup>In späteren Kapiteln, insbesondere in Abschnitt 4.2.2, werden wir hierauf noch genauer eingehen.

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

*throw*-  
Anweisung

```
throw Ausdruck ;
```

Die *throw*-Anweisung terminiert immer abrupt. Der Ausdruck muss zu einem Ausnahmeobjekt auswerten. Als Beispiel nehmen wir an, dass wir eine Ausnahmebehandlung für Überläufe bei *int*-Additionen entwickeln möchten (in Java werden solche Überläufe normalerweise nicht behandelt).

Dazu deklarieren wir eine Ausnahmeklasse *Ueberlauf*. Alle selbstdefinierten Ausnahmeklassen müssen in Java direkt oder indirekt der Klasse *Exception* untergeordnet sein. Daher definieren wir die Klasse *Ueberlauf* wie folgt:

```
class Ueberlauf extends Exception {}
```

Die Klasse *Ueberlauf* leitet dabei die Deklaration der Klasse *Ueberlauf* ein. *extends Exception* bedeutet, dass *Ueberlauf* als eine *Exception* untergeordnete Klasse deklariert wird. *Ueberlauf* erbt dadurch alle Methoden und Attribute von *Exception*. Die Deklaration eigener Ausnahmetypen wird in Kap. 4 genauer behandelt. Der in Zeile 8 der Abbildung 1.11 vorkommende Ausdruck *new Ueberlauf()* erzeugt ein Ausnahmeobjekt vom Typ *Ueberlauf*. Um die Anwendung der *throw*-Anweisung zu demonstrieren, haben wir im obigen Programmfragment die Division durch eine Addition im Typ *long* ersetzt. Abbildung 1.11 zeigt das Resultat.

```
(1) String[] argf = {"2147483640", "3450336"};
(2) long maxint = 2147483647L;
(3) try {
(4)     int m, n, ergebnis ;
(5)     m = Integer.parseInt( argf[0] );
(6)     n = Integer.parseInt( argf[1] );
(7)     long aux = (long)m + (long)n;
(8)     if( aux > maxint ) throw new Ueberlauf();
(9)     ergebnis = (int)aux ;
(10) } catch ( IndexOutOfBoundsException e ) {
(11)     System.out.println("argf falsch initialisiert");
(12) } catch ( NumberFormatException e ) {
(13)     System.out.println(
(14)         "Element in argf keine int-Konstante");
(15) } catch ( Ueberlauf e ) {
(16)     System.out.println("Ueberlauf aufgetreten");
(17) }
```

Abbildung 1.11: Demonstration der *throw*-Anweisung

Insgesamt kann der Programmierer durch geeigneten Einsatz der *try*- und der *throw*-Anweisung verbunden mit der Definition eigener Ausnahmetypen eine flexible Ausnahmebehandlung realisieren.

### 1.3.2 Objektorientierte Programmierung mit Java

Dieser Abschnitt führt in die programmiersprachliche Realisierung objektorientierter Konzepte ein. Anhand eines kleinen Beispiels demonstriert er, wie die zentralen Aspekte des objektorientierten Grundmodells, das in Abschn. 1.2.3 vorgestellt wurde, mit Hilfe von Java umgesetzt werden können. Insgesamt verfolgt dieser Abschnitt die folgenden Ziele:

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

Das verwendete Programmierbeispiel lehnt sich an die Personenstruktur von Abb. 1.4 an (s. S. 22). Wir definieren Personen- und Studenten-Objekte und statten sie zur Illustration mit einfachen Methoden aus. Wir zeigen, wie das Erzeugen von Objekten realisiert und wie mit Objekten programmiert werden kann.

#### 1.3.2.1 Objekte, Klassen, Methoden, Konstruktoren

Objekte besitzen einen Zustand und Methoden, mit denen sie auf Nachrichten reagieren (vgl. Abb. 1.3, S. 21). Programmtechnisch wird der Zustand durch mehrere Attribute realisiert, also objektlokale Variablen. Solche objektlokalen Variablen werden in Java im Rahmen von Klassen deklariert. Bei der Erzeugung eines Objekts werden diese Variablen dann initialisiert. Im Gegensatz zu z.B. C++ speichern Attribute in Java Objektreferenzen (Zeiger auf Objekte) und nicht die Objekte selbst.

Personen-Objekte mit den in Abb. 1.4 angegebenen Attributen `name` und `geburtsdatum`<sup>11</sup> werden dann spezifiziert durch eine Klasse `Person`:

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

Objekte des Typs<sup>12</sup> `Person` kann man durch Aufruf des *default-Konstruktors* `new Person()` erzeugen. Durch diesen Aufruf wird ein Speicherbereich für das neue Objekt alloziert und die Attribute des Objekts

---

<sup>11</sup>Achtung: in diesem Java-Beispiel kann man direkt auf diese Attribute zugreifen. Bei Abb. 1.4 wurden statt dessen Zugriffsmethoden angenommen, was man in Java auch realisieren könnte, aber hier das Beispiel zu unübersichtlich macht.

<sup>12</sup>Klassen sind in Java spezielle Typen.

mit Standardwerten initialisiert. Möchte man die Initialisierungswerte selbst festlegen, muss man der Klassendeklaration einen selbstgeschriebenen *Konstruktor* hinzufügen:

*Konstruktor*

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

    Person( String n, int gd ) {
        name = n;
        geburtsdatum = gd;
    }
};
```

Der Name eines Konstruktors muss mit dem Namen der Klasse übereinstimmen, zu der er gehört und kann Parameter besitzen, die zur Initialisierung der Attribute verwendet werden können (s.o.). Näheres zu Konstruktoren finden Sie in Abschnitt 2.1.2.

Wir statten Personen-Objekte nun mit zwei Methoden aus: Erhält eine Person die Nachricht `drucken`, soll sie ihren Namen und ihr Geburtsdatum drucken; erhält eine Person die Nachricht `hat_geburtstag` mit einem Datum als Parameter, soll sie prüfen, ob sie an dem Datum Geburtstag hat (Datumsangaben werden als int-Zahl im Format JJJJMMTT codiert). Die bisherige Typdeklaration von `Person` wird also erweitert um die Methoden `drucken` und `hat_geburtstag`:

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

    Person( String n, int gd ) {
        name = n;
        geburtsdatum = gd;
    }

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

    boolean hat_geburtstag ( int datum ) {
        return (this.geburtsdatum%10000) == (datum%10000);
    }
}
```

Die Methode `drucken` druckt die beiden Attribute `name` und `geburtsdatum` des Personen-Objekts aus, auf dem sie aufgerufen wird.

*this-Objekt*

Dieses Personen-Objekt wird der Methode bei ihrem Aufruf als impliziter Parameter mitgegeben. Dieser Parameter wird häufig als *this-Objekt* oder *self-Objekt* bezeichnet.

*formaler Parameter*

Wird einer Methode neben dem impliziten *this*-Parameter wie bei `hat_geburtstag` noch ein weiterer Parameter übergeben (hier `datum`), so wird dieser zur besseren Unterscheidung auch *formaler Parameter* genannt. In `hat_geburtstag` wird dieser mit der Monats- und Tagesangabe des entsprechenden Personen-Objekts verglichen.

### 1.3.2.2 Spezialisierung und Vererbung

Gemäß Abb. 1.4, S. 22, ist ein Student eine Person mit zwei zusätzlichen Attributen. Studenten sind also spezielle Personen. Es wäre demnach wünschenswert, die Implementierung von Studenten-Objekten durch geeignete Erweiterung und Modifikation der Implementierung von Personen-Objekten zu erhalten. Da die Methode `hat_geburtstag` auch für Studenten-Objekte korrekt funktioniert, können wir sie übernehmen.

Objektorientierte Programmiersprachen wie Java unterstützen es, Klassen wie gewünscht zu erweitern. Beispielsweise ist die folgende Klasse `Student` eine Erweiterung der Klasse `Person`: `Student extends Person`. Sie erbt alle Attribute und Methoden von `Person` – im Beispiel die Attribute `name` und `geburtsdatum` und die Methode `hat_geburtstag`. Die Methoden und Konstruktoren der Klasse, von der geerbt wird, können mit dem Schlüsselwort `super` angesprochen werden:

```
class Student extends Person {
    int matrikelnr;
    int semester;

    Student( String n, int gd, int mnr, int sem ) {
        super( n, gd );
        matrikelnr = mnr;
        semester = sem;
    }
    void drucken() {
        super.drucken();
        System.out.println( "Matrikelnr: " + matrikelnr );
        System.out.println( "Semesterzahl: " + semester );
    }
}
```

Anhand des obigen Beispiels lassen sich die Aspekte erkennen, die für die Vererbung wichtig sind. Der speziellere Typ (hier `Student`) besitzt alle Attribute des allgemeineren Typs (hier `Person`). Er kann auf die gleichen Nachrichten reagieren (hier auf `drucken` und `hat_geburtstag`), ggf. auch auf

zusätzliche Nachrichten (hier nicht demonstriert). Er kann also die Attribute und die Typen der Nachrichten vom allgemeineren Typ erben. Er kann auch die Methoden erben, wie wir am Beispiel von `hat_geburtstag` gesehen haben. In vielen Fällen reichen die Methoden des allgemeineren Typs aber nicht aus, wie das Beispiel von `drucken` zeigt. Sie müssen durch eine spezielle Variante ersetzt werden. In solchen Fällen spricht man von *Überschreiben* der Methode des allgemeineren Typs. Allerdings ist es häufig sinnvoll die überschriebene Methode zur Implementierung der überschreibenden Methode heranzuziehen. In unserem Beispiel wird in der Methode `drucken` der Klasse `Student` durch den Aufruf von

*Überschreiben*

```
super.drucken();
```

die Methode `drucken` der Klasse `Person` aufgerufen und dazu genutzt, die Attribute `name` und `geburtsdatum` auszudrucken. Der Aufruf von

```
super( n, gd );
```

im Konstruktor von `Student` bewirkt, dass der Konstruktor der Klasse `Person` aufgerufen wird.

### 1.3.2.3 Subtyping und dynamisches Binden

Im Prinzip können Studenten-Objekte an allen Programmstellen verwendet werden, an denen Personen-Objekte zulässig sind; denn sie unterstützen alle Operationen, die man auf Personen-Objekte anwenden kann, wie Attributzugriff und Methodenaufruf. In der objektorientierten Programmierung gestattet man es daher, Objekte von spezielleren Typen überall dort zu verwenden, wo Objekte von allgemeineren Typen zulässig sind. Ist  $S$  ein speziellerer Typ als  $T$ , so wird  $S$  auch als *Subtyp* von  $T$  bezeichnet; in diesem Sinne ist `Student` ein Subtyp von `Person`.

*Subtyp*

Der entscheidende Vorteil von Subtyping ist, dass wir einen Algorithmus, der für einen Typ formuliert ist, für Objekte aller Subtypen verwenden können. Um dies an unserem Beispiel zu demonstrieren, greifen wir eine Variante des Problems aus Abschn. 1.2.3, S. 23, auf: Wir wollen alle Elemente eines Felds mit Komponenten vom Typ `Person` drucken. Das folgende Programmfragment zeigt, wie einfach das mit objektorientierten Techniken geht:

```
int i;
Person[] pf = new Person[3];
pf[0] = new Person( "Meyer", 19631007 );
pf[1] = new Student( "Mueller", 19641223, 6758475, 5 );
pf[2] = new Student( "Planck", 18580423, 3454545, 47 );

for( i = 0; i < 3; i = i + 1 ) {
    pf[i].drucken();
}
```



In der dritten bis fünften Zeile wird das Feld mit einem Personen-Objekt und zwei Studenten-Objekten initialisiert. In der for-Schleife braucht nur die Druckmethode für jedes Element aufgerufen zu werden. Ist das Element eine Person, wird die Methode `drucken` der Klasse `Person` ausgeführt; ist das Element ein Student, wird die Methode `drucken` der Klasse `Student` ausgeführt. Drei Aspekte sind dabei bemerkenswert:

1. Eine Fallunterscheidung an der Aufrufstelle, wie sie in der Pascal-Version von S. 23 nötig war, kann entfallen.
2. Die Schleife braucht nicht geändert zu werden, wenn das Programm um neue Personenarten, z.B. Angestellte, erweitert wird.
3. Der Aufruf `pf[i].drucken()` führt im Allgemeinen zur Ausführung unterschiedlicher Methoden.

Der letzte Punkt bedeutet, dass die Zuordnung von Aufrufstelle und ausgeführter Methode nicht mehr vom Übersetzer vorgenommen werden kann, sondern erst zur Laufzeit erfolgen kann. Da man alle Dinge, die zur Laufzeit geschehen als *dynamisch* bezeichnet, spricht man von *dynamischer Bindung* der Methoden. Alle Dinge, die zur Übersetzungszeit behandelt werden können, werden als *statisch* bezeichnet.

*dynamische  
Bindung*

### 1.3.3 Aufbau eines Java-Programms

Aus einer vereinfachten Sicht eines Programmierers besteht ein Java-Programm aus einer oder mehreren von ihm entwickelten Klassen<sup>13</sup>, von denen eine Klasse eine sogenannte *main-Methode* besitzen muss, die den Programmeinstiegspunkt festlegt:

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

Die Klasse mit dem Programmeinstiegspunkt muss in einer Datei enthalten sein, die die Dateiendung `.java` besitzt. Am Besten gibt man dieser Datei den Namen der Klasse mit dem Programmeinstiegspunkt, in unserem Fall also `Main.java`.

Die *main-Methode* hat stets den oben beschriebenen Aufbau: sie muss den Namen `main` besitzen, darf keine Werte zurückliefern (angegeben durch

<sup>13</sup>Im Absatz „Objektorientierte Programme“ von Abschnitt 2.1.2 „Klassen beschreiben Objekte“ wird eine differenziertere Sicht auf ein Java-Programm beschrieben, die auch Bibliotheksklassen einbezieht.

`void`) und nur einen Parameter vom Typ `String[]` besitzen. Dieser Parameter dient dazu, Argumente aufzunehmen, die dem Java-Programm beim Aufruf mitgegeben werden (s.u.).

Bei Verwendung des J2SE Development Kits (vgl. Einleitung <sup>14</sup>) kann man die Klasse `Main` mit dem Kommando<sup>15</sup>

```
javac Main.java
```

übersetzen. Die Übersetzung liefert eine Datei `Main.class`, die man mit

```
java Main
```

ausführen kann. Beim Aufruf ist auf die korrekte Groß-/Kleinschreibung des Klassennamens zu achten.

Wie bereits oben erwähnt, können einem Programmaufruf Argumente, auch Programmparameter genannt, mitgegeben werden. Diese sind durch Leerzeichen voneinander zu trennen. Der folgende Aufruf demonstriert dieses Vorgehen:

```
java Main Hello World!
```

Auf solche Argumente kann man im Programm über den Parameter `args` der Methode `main` zugreifen: `args[0]` liefert das erste Argument, `args[1]` das zweite usw. Dies demonstriert das folgende vollständige Java-Programm, das die ersten beiden Argumente und die Zeichenreihe „Na endlich, das erste Programm mit Argumenten laeuft!“ ausgibt:

```
class Main {
    public static void main( String[] args )
    {
        if (args.length > 1) {
            System.out.print(args[0]);
            System.out.print(" ");
            System.out.print(args[1]);
            System.out.println();
        }
        System.out.println
            ("Na endlich, das erste Programm mit Argumenten laeuft!");
    }
}
```

---

<sup>14</sup>J2SE Development Kit ist der ab Version 5.0 verwendete Name für die Entwicklungsumgebung, der die früher verwendeten Namen Java 2 Software Development Kit und Java Development Kit ersetzt.

<sup>15</sup>In Windows können solche Kommandos über die Windows-Eingabeaufforderung eingegeben werden.

Ein etwas größeres Programm besteht aus den uns bereits bekannten Klassen **Person** und **Student** sowie der Klasse **Test**, die die **main**-Methode enthält.

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

    Person( String n, int gd ) {
        name = n;
        geburtsdatum = gd;
    }
    void drucken() {
        System.out.println("Name: "+ this.name);
        System.out.println("Geburtsdatum: "+ this.geburtsdatum);
    }
    boolean hat_geburtstag ( int datum ) {
        return (this.geburtsdatum%10000) == (datum%10000);
    }
}

class Student extends Person {
    int matrikelnr;
    int semester;

    Student( String n, int gd, int mnr, int sem ) {
        super( n, gd );
        matrikelnr = mnr;
        semester = sem;
    }
    void drucken() {
        super.drucken();
        System.out.println( "Matrikelnr: " + matrikelnr );
        System.out.println( "Semesterzahl: " + semester );
    }
}

class Test {
    public static void main( String[] args ) {
        int i;
        Person[] pf = new Person[3];
        pf[0] = new Person( "Meyer", 19631007 );
        pf[1] = new Student( "Mueller", 19641223, 6758475, 5 );
        pf[2] = new Student( "Planck", 18580423, 3454545, 47 );

        for( i = 0; i < 3; i = i + 1 ) {
```

```
        pf[i].drucken();  
    }  
}  
}
```

### 1.3.4 Objektorientierte Sprachen im Überblick

Objektorientierte Sprachen unterscheiden sich darin, wie sie das objektorientierte Grundmodell durch Sprachkonstrukte unterstützen und wie sie die objektorientierten Konzepte mit anderen Konzepten verbinden. Mittlerweile gibt es etliche Sprachen, die objektorientierte Aspekte unterstützen. Viele von ihnen sind durch Erweiterung existierender Sprachen entstanden, beispielsweise C++ und Objective C (als Erweiterung von C), CLOS (als Erweiterung von Common Lisp), Ada95, Object Pascal<sup>16</sup> und Modula-3. Andere Sprachen wurden speziell für die objektorientierte Programmierung entwickelt, beispielsweise Simula, Smalltalk, Eiffel, BETA und Java.

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

**Gemeinsamkeiten.** In fast allen objektorientierten Sprachen wird ein Objekt als ein Verbund von Variablen realisiert, d.h. so, wie wir es in Abschn. 1.3.2 erläutert haben. Allerdings werden die Methodenimplementierungstechnisch in den meisten Fällen nicht den Objekten zugeordnet, sondern den Klassen. Eine verbreitete Gemeinsamkeit objektorientierter Sprachen ist die *synchrone* Kommunikation, d.h. der Aufrufer einer Methode kann erst fortfahren, wenn die Methode terminiert hat. Mit dem Vokabular des objektorientierten Grundmodells formuliert, heißt das, dass Senderobjekte nach dem Verschicken einer Nachricht warten müssen, bis die zugehörige Methode vom Empfänger bearbeitet wurde; erst nach der Bearbeitung und ggf. nach Empfang eines Ergebnisses kann der Sender seine Ausführung fortsetzen. Das objektorientierte Grundmodell ließe auch andere Kommunikationsarten

---

<sup>16</sup>Object Pascal liegt u.a. der verbreiteten Programmentwicklungsumgebung Delphi zugrunde.

zu (vgl. Kap. 7). Beispiele dafür findet man vor allem bei objektorientierten Sprachen und Systemen zur Realisierung verteilter Anwendungen.

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

- **Objektbeschreibung:** In den meisten Sprachen werden Objekte durch sogenannte Klassendeklarationen beschrieben. Andere Sprachen verzichten auf Klassen und bieten Konstrukte an, mit denen man existierende Objekte während der Programmlaufzeit klonen kann und dann Attribute und Methoden hinzufügen bzw. entfernen kann. Derartige Sprachen nennt man *prototypbasiert* (Beispiel: die Sprache Self).
- **Reflexion:** Sprachen unterscheiden sich darin, ob Klassen und Methoden auch als Objekte realisiert sind, d.h. einen Zustand besitzen und Empfänger und Sender von Nachrichten sein können. Beispielsweise sind in Smalltalk Klassen Objekte, und in BETA sind Methoden als Objekte modelliert.
- **Typsysteme:** Die Typsysteme objektorientierter Sprachen sind sehr verschieden. Am einen Ende der Skala stehen untypisierte Sprachen (z.B. Smalltalk), am anderen Ende Sprachen mit strenger Typprüfung, Subtyping und Generizität (z.B. Java und Eiffel).
- **Vererbung:** Auch bei der Vererbung von Programmteilen zwischen Klassen gibt es wichtige Unterschiede. In vielen Sprachen, z.B. in Java oder C#, kann eine Klasse nur von *einer* anderen Klasse erben (Einfachvererbung), andere Sprachen ermöglichen Mehrfachvererbung (z.B. CLOS und C++). Meist sind Vererbung und Subtyping eng aneinander gekoppelt. Es gibt aber auch Sprachen, die diese Konzepte sauber voneinander trennen (beispielsweise Sather). Variationen gibt es auch dabei, was für Programmteile vererbt werden können.
- **Spezialisierung:** Objektorientierte Sprachen bieten Sprachkonstrukte an, um geerbte Methoden zu spezialisieren. Diese Sprachkonstrukte unterscheiden sich zum Teil erheblich voneinander.
- **Kapselung:** Kapselungskonstrukte sollen Teile der Implementierung vor dem Benutzer verbergen. Die meisten OO-Sprachen unterstützen Kapselung in der einen oder anderen Weise. Allerdings gibt es auch Sprachen, die Kapselung nur auf Modulebene und nicht für Klassen unterstützen (z.B. BETA).
- **Strukturierungskonstrukte:** Ähnlich der uneinheitlichen Situation bei der Kapselung gibt es eine Vielfalt von Lösungen zur Strukturierung einer Menge von Klassen. Einige Sprachen sehen dafür Modulkonzepte vor (z.B. Modula-3), andere benutzen die Schachtelung von Klassen zur Strukturierung.

- Implementierungsaspekte: Ein wichtiger Implementierungsaspekt objektorientierter Programmiersprachen ist die Frage, wer den Speicher für die Objekte verwaltet. Beispielsweise ist in C++ der Programmierer dafür zuständig, während Java eine automatische Speicherverwaltung anbietet. Ein anderer Aspekt ist die Ausrichtung der Sprache auf die Implementierungstechnik. Sprachen, die für Interpretation entworfen sind, sind meist besser für dynamisches Laden von Klassen und ähnliche Techniken geeignet als Sprachen, die für Übersetzung in Maschinensprache ausgelegt sind.

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

## 1.4 Aufbau und thematische Einordnung des Kurses

Dieser Abschnitt erläutert den Aufbau des Kurses und bietet eine kurze Übersicht über andere einführende Bücher im thematischen Umfeld.

**Aufbau.** Dieser Kurs lässt sich von den objektorientierten Konzepten leiten und nimmt diese als Ausgangspunkt für die Behandlung der sprachlichen Realisierung. Dieses Vorgehen spiegelt sich in der Struktur des Kurses wider. Jedes der sechs Hauptkapitel stellt einen zentralen Aspekt objektorientierter Programmierung vor:

- Kapitel 2: Objekte, Klassen, Kapselung. Dieses Kapitel erläutert, was Objekte sind, wie man sie mit Klassen beschreiben kann, wozu Kapselung dient, wie Klassen strukturiert werden können und welche Beziehungen es zwischen Klassen gibt.
- Kapitel 3: Vererbung und Subtyping. Ausgehend vom allgemeinen Konzept der Klassifikation werden die zentralen Begriffe Vererbung und Subtyping erläutert und ihre programmiersprachliche Realisierung beschrieben.
- Kapitel 4: Bausteine für objektorientierte Programme. Dieses Kapitel behandelt die Nutzung objektorientierter Techniken für die Entwicklung wiederverwendbarer Programmbausteine. In diesem Zusammenhang werden auch typische Klassen aus der Java-Bibliothek vorgestellt.

- Kapitel 5: Objektorientierte Programmgerüste. Oft benötigt man ein komplexes Zusammenspiel mehrerer erweiterbarer Klassen, um eine softwaretechnische Aufgabenstellung zu lösen. Programmgerüste bilden die Basis, mit der häufig vorkommende, ähnlich gelagerte Aufgabenstellungen bewältigt werden können, beispielsweise die Realisierung graphischer Bedienoberflächen. In diesem Kapitel erläutern wir das Abstract Window Toolkit von Java als Beispiel für ein objektorientiertes Programmgerüst.
- Kapitel 6: Parallelität in objektorientierten Programmen. Wie bereits auf Seite 20 erwähnt, unterstützen gängige objektorientierte Programmiersprachen i.A. keine inhärente Parallelität. Statt dessen muss Parallelität explizit programmiert werden. Dieses Kapitel behandelt die Realisierung von expliziter Parallelität in objektorientierten Programmen anhand des Thread-Konzepts von Java und führt in das zugehörige objektorientierte Monitorkonzept ein.
- Kapitel 7: Verteilte Programmierung mit Objekten. Die Realisierung verteilter Anwendungen wird in der Praxis immer bedeutsamer. Die objektorientierte Programmierung leistet dazu einen wichtigen Beitrag. Dieses Kapitel stellt die dafür entwickelten Techniken und deren sprachliche Unterstützung vor. Dabei werden wir uns auf Anwendungen im Internet konzentrieren.

Diese Struktur lässt sich wie folgt zusammenfassen: Kapitel 2 und 3 stellen die Grundkonzepte sequentieller objektorientierter Programmierung vor. Kapitel 4 und 5 zeigen, wie diese Konzepte für die Entwicklung von Programmbibliotheken und wiederverwendbaren Programmgerüsten genutzt werden können. Kapitel 6 und 7 behandeln die quasi-parallele objektorientierte Programmierung. Am Ende des Kurses bringt Kapitel 8 eine Zusammenfassung, behandelt Varianten bei der Realisierung objektorientierter Konzepte und bietet einen kurzen Ausblick.

Die behandelten Konzepte, Techniken und Sprachkonstrukte werden mit Hilfe von Beispielen illustriert. Um das Zusammenwirken der Techniken über die Kapitelgrenzen hinweg zu demonstrieren, wird schrittweise ein rudimentärer Internet-Browser entwickelt und besprochen.

**Einordnung in die Literatur.** Erlaubt man sich eine vereinfachende Betrachtungsweise, dann kann man die Literatur zur Objektorientierung in vier Bereiche einteilen. Dieser Absatz stellt die Bereiche kurz vor und gibt entsprechende Literaturangaben.

Der erste Bereich beschäftigt sich mit objektorientiertem Software-Engineering, insbesondere mit der objektorientierten Analyse und dem objektorientierten Entwurf. Zentrale Fragestellungen sind in diesem Zusammen-



hang: Wie entwickelt man ausgehend von einer geeigneten Anforderungsanalyse einen objektorientierten Systementwurf? Wie können objektorientierte Modelle und Techniken dafür nutzbringend eingesetzt werden? Eine gute und übersichtliche Einführung in dieses Gebiet bietet [HS97]. Insbesondere enthält das Buch ein ausführliches, annotiertes Literaturverzeichnis und eine Übersicht über einschlägige Konferenzen und Zeitschriften zur Objektorientierung.

Als zweiten Bereich betrachten wir die objektorientierte Programmierung, d.h. die Realisierung objektorientierter Programme ausgehend von einem allgemeinen oder objektorientierten Softwareentwurf. Sofern das nicht von Anfang an geschehen ist, muss dazu der Entwurf in Richtung auf eine objektorientierte Programmstruktur hin verfeinert werden. Ergebnis einer solchen Verfeinerung ist typischerweise eine Beschreibung, die angibt, aus welchen Klassen das zu entwickelnde System aufgebaut sein soll, welche existierenden Klassen dafür wiederverwendet werden können und welche Klassen neu zu implementieren sind. Dieser so verfeinerte Entwurf ist dann – meist unter Verwendung objektorientierter Programmiersprachen – zu codieren. Für den ersten Schritt benötigt man die Kenntnis objektorientierter Programmierkonzepte, für den zweiten Schritt ist die Beherrschung objektorientierter Programmiersprachen vonnöten. Eine exemplarisch gehaltene Darstellung des Zusammenhangs zwischen objektorientiertem Entwurf und objektorientierter Programmierung bietet [Cox86].

Der vorliegende Kurstext erläutert die Konzepte objektorientierter Programmierung und deren Umsetzung in der Sprache Java. Ähnlich gelagert ist das Buch [Mey00] von B. Meyer. Es führt die Konzepte allerdings anhand der Programmiersprache Eiffel ein und geht darüberhinaus auch auf Aspekte des objektorientierten Entwurfs ein. Desweiteren bietet das Buch [Bud01] von T. Budd eine sehr lesenswerte Einführung in verschiedene Aspekte objektorientierter Programmierung. Dabei wird bewußt mit Beispielen in unterschiedlichen Sprachen gearbeitet. Beide Bücher verzichten aber auf eine Behandlung von Programmgerüsten und von paralleler bzw. verteilter Programmierung.

In einem dritten Bereich der Objektorientierung siedeln wir Bücher an, die sich im Wesentlichen nur mit einer objektorientierten Programmiersprache beschäftigen. Selbstverständlich ist die Grenze zum zweiten Bereich fließend. Bücher, die auch konzeptionell interessante Aspekte enthalten, sind beispielsweise zu Beta [MMPN93], zu C++ [Lip91] und zu Java [HC97, HC98].

Orthogonal zu den drei genannten Bereichen liegen Arbeiten zur theoretischen Fundierung der Objektorientierung. Das Buch [AC96] entwickelt eine Theorie für Objekte ausgehend von verschiedenen Kalkülen und liefert damit zentrale programmiersprachliche Grundlagen. Eine gute Übersicht über den Stand der Technik von formalen Entwicklungsmethoden für objektorientierte Software bietet das Buch [GK96].

## Selbsttestaufgaben

### Aufgabe 1: Der intelligente Kühlschrank

Ein intelligenter Kühlschrank kommuniziert mit seiner Umwelt und mit sich selbst.

- Er testet regelmäßig seine Komponenten auf Funktionstüchtigkeit. Bei Defekten sendet er eine Nachricht an den Reparaturdienst und / oder informiert seinen Besitzer.
- Er prüft ständig, welche der Nahrungsmittel und sonstigen Gegenstände, die sein Besitzer in ihm aufheben will, vorhanden sind.
- Wenn ein Nahrungsmittel zu Ende geht, sendet er eine Nachricht an das Geschäft, das dieses Nahrungsmittel verkauft.
- Der Hausroboter sendet Nachrichten an den Kühlschrank, wenn eine Lieferung angekommen ist.

Entwickeln Sie ein objektorientiertes Modell eines intelligenten Kühlschranks, indem Sie die Gegenstände seiner Welt angeben und die Methoden, die der Kühlschrank und diese Gegenstände besitzen müssen, um zu kommunizieren und ihre Aufgaben zu erledigen.

### Aufgabe 2: Erste Schritte in Java

Java-Programme werden normalerweise von einer sogenannten *virtuellen Maschine* ausgeführt. Ein Java-Programm wird mit einem Übersetzer in einen Zwischencode (Bytecode) übersetzt. Eine virtuelle Maschine ist ein Programm, das Bytecode ausführen kann. Für verschiedene Rechnerarchitekturen und Betriebssysteme existieren virtuelle Maschinen, die diesen Zwischencode ausführen können. Der Bytecode ist unabhängig von der jeweiligen Rechnerumgebung und kann so auf jedem Rechner ausgeführt werden, für den eine virtuelle Maschine existiert.

Im Folgenden sollen Sie das Java Software Development Kit installieren und kleine Java-Programme erstellen und ausführen, um sich mit der Syntax der Sprache, der Bedienung des Übersetzers und der virtuellen Maschine vertraut zu machen.

#### a) Installation der Java-Entwicklungsumgebung (Erfahrung)

Installieren Sie die Java-Entwicklungsumgebung JDK 5.0. Einen Link zum Downloaden der Installationssoftware können Sie auf unseren Kurswebseiten finden. Dort finden Sie auch Hinweise zur Installation, ebenso wie einen Link zum PC-Tutorial, das u.A. wertvolle Hinweise zur Installation des JDK enthält, zum Setzen von Umgebungsvariablen unter Windows etc.

**b) Das erste Java-Programm****(Erfahrung)**

1. Legen Sie ein Verzeichnis an, in dem Sie Ihre Java-Klassen ablegen wollen.
2. Erstellen Sie mit einem Editor Ihres Systems eine Datei namens `Start.java`, die folgenden Inhalt hat:

```
public class Start {  
    public static void main(String[] args) {  
        System.out.println("Mein erstes Java-Programm");  
    }  
}
```

3. Erzeugen Sie mit dem Java-Übersetzer `javac` aus dem eben erstellten Programmtext eine `.class`-Datei: `'javac Start.java'`. Die `.class` Datei enthält den oben beschriebenen Bytecode für die Klasse `Start`.
4. Jede `.class`-Datei, deren zugehörige Klasse eine `main`-Methode enthält, kann mit der virtuellen Maschine von Java ausgeführt werden: Geben Sie den Befehl `'java Start'` ein<sup>17</sup>. Der Code des eben beschriebenen Programms wird jetzt ausgeführt.
5. Ergänzen Sie jetzt Ihr `Start`-Programm in der folgenden Art und Weise:

```
public class MeinStart {  
    public static void main(String[] args) {  
        String Nachname;  
        String Vorname;  
        Vorname = args[0];  
        Nachname = args[1];  
        System.out.println("Mein erstes Java-Programm");  
        System.out.println("geschrieben von " + Vorname +  
                           " " + Nachname );  
    }  
}
```

Speichern Sie es unter `MeinStart.java` ab. Erzeugen Sie wieder eine `.class`-Datei. Geben Sie den Befehl

`'java MeinStart MeinVorname MeinNachname'` ein.

---

<sup>17</sup>Beim Starten wird die Dateierweiterung `.class` weggelassen.

**Aufgabe 3: Implizite Typkonvertierungen**

Nennen Sie im folgenden Java-Programm alle Stellen, an denen implizite Typkonvertierungen vorkommen und begründen Sie Ihre Aussage.

```
class Main {  
    public static void main( String[] args ) {  
(1)  int i = 10;  
(2)  long l = 55567843L;  
(3)  byte by = 15;  
(4)  boolean b = true;  
(5)  double d = 1.25;  
(6)  l = i;  
(7)  d = l;  
(8)  by = i;  
(9)  l = l + by;  
(10) by = by - b;  
(11) d = (l / i) * 20;  
    }  
}
```

Das Programm enthält auch Zuweisungen und Operationen, die nicht erlaubt sind. Nennen Sie auch solche Stellen und begründen Sie Ihre Aussage.

**Aufgabe 4: Schleifen**

Erstellen Sie ein einfaches Java-Programm, das alle beim Programmaufruf mitgegebenen Programmparameter auf der Standardausgabe jeweils in einer eigenen Zeile ausgibt. Verwenden Sie zur Ausgabe alle im Kurstext eingeführten Schleifenkonstrukte.

**Aufgabe 5: Kontrollstrukturen**

Schreiben Sie ein Java-Programm, das die ersten zwei Programmparameter in Integerzahlen umwandelt und von beiden Zahlen den größten gemeinsamen Teiler bestimmt. Prüfen Sie dann, ob der ermittelte größte gemeinsame Teiler einer der Zahlen von 1 bis 4 ist. Verwenden Sie für diesen Test eine `switch`-Anweisung und geben Sie in dem jeweils zutreffenden Fall eine entsprechende Meldung aus. Verwenden Sie den `default`-Fall, wenn der größte gemeinsame Teiler größer als 4 ist.

**Aufgabe 6: Sortieren eines Feldes**

Gegeben sei das folgende noch mit einigen syntaktischen Fehlern behaftete Programmfragment, das wir in dieser Aufgabe korrigieren und vervollständigen wollen.

```

class Sortieren {
    public static void main (String[] args) {
        // Ein double Feld erzeugen, das genauso gross ist wie das
        // args-Feld
        double[] feld = new double[args.length]

        // alle Zahlen, die in args als Strings
        // vorliegen, in double-Werte umwandeln
        // und in das Feld feld eintragen
        for(int i = 0; i < args.length, i = i + 1 {
            feld[i] = Double.parseDouble(args(i));
        }

        // Hier Programmcode zum Sortieren einfüegen

        // Hier Programmcode zur Bestimmung und
        // Ausgabe des grössten Elements einfüegen

        // den Inhalt den Feldes feld ausgeben
        for(int i := 0; i < args.length; i = i + 1) {
            System.out.println(i + ". " + feld[i]);
        }
    }
}

```

Die main-Methode eines Programms bekommt beim Aufruf des Programms angegebene Parameter im Feld `args` vom Typ `String` übergeben.

Beispiel: `java Sortieren 1.2 3.56 2.9 -23.4 3.1415926`

Die Korrektur der syntaktischen Fehler vorausgesetzt, wandelt das obige Programm die der main-Methode als `String`-Feld übergebenen Parameter mit der Methode `Double.parseDouble()` in ein Feld mit `double`-Werten um. Diese werden dann ausgegeben.

### Aufgaben:

1. Das obige Programm enthält noch einige Syntaxfehler. Markieren Sie zunächst alle Fehlerstellen möglichst genau zusammen mit einer (knappen) Fehlerbeschreibung. Korrigieren Sie anschließend die erkannten Syntaxfehler und überprüfen Sie, ob Sie tatsächlich alle Fehler gefunden haben und Ihr korrigiertes Programm vom Java-Übersetzer akzeptiert wird. Korrigieren Sie ggf. die noch verbliebenen Fehler, bevor Sie mit der zweiten Teilaufgabe und der Erweiterung des Programms beginnen<sup>18</sup>.

<sup>18</sup>Eine Umleitung der Fehlermeldungen des Java-Compilers von der Bildschirmausgabe in eine Datei mit Namen `error` können Sie durch folgenden Aufruf erreichen: `javac -Xstdout error Sortieren.java`

2. Erweitern Sie das korrigierte Programm an den gekennzeichneten Stellen um Programmcode, der das Feld mit den als Parameter übergebenen double-Werten absteigend sortiert und anschließend das größte Element bestimmt. Testen Sie Ihr Programm mit verschiedenen Eingaben.

### Aufgabe 7: Ausnahmebehandlung (Brandschutzübung)

Um die jährliche Brandschutzübung an der FernUniversität für die Informatiker interessanter zu gestalten, hat die Hager Feuerwehr das Motto der diesjährigen Übung mit einem Java-Programm codiert. Da für die Feuerwehr tägliche Alarmer und Ausnahmesituationen die Regel sind, hat sie dabei reichlich Gebrauch von Exceptions gemacht.

1. Wie heißt das Motto, das von dem unten angegebenen Programm bei Ausführung ausgegeben wird?
2. Beschreiben Sie detailliert, was bei der Ausführung des Programms passiert, d.h. welche Exceptions auftreten und wo diese abgefangen und behandelt werden!<sup>19</sup>

```
public class Alarm_Alarm {

    public static void main(String[] args) {
        try {
            try {
                int i = 7 % 5;
                if ( (i / (i % 2)) == 1) throw new Exception();
                System.out.println("leichtsinnig");
            }

            catch (Exception e) {
                System.out.println("man");
                try {
                    if ( (7 % 6 / (7 % 6 % 2)) == 1) throw new Exception();
                    System.out.println("leichtsinnig");
                }
                catch (Exception u) {
                    System.out.println("spielt");
                }
            }

            System.out.println("nicht");
        }
    }
}
```

---

<sup>19</sup>Beachten Sie bei der Lösung, dass die Klasse `ArithmeticException` Subtyp der Klasse `Exception` ist.

```
try {
    int i = true & false ? 0 : 1;
    switch (i) {
        case 1:
            System.out.println("mit");
        default:
            throw new Exception();
    }
}
catch (ArithmeticException e) {
    System.out.println("Streichhoelzern");
}
catch (Exception e) {
    System.out.println("Feuer");
}

finally {
    int i = false && true ? 0 : 2;
    switch (i) {
        case 1:
            System.out.println("mit");
        default:
            throw new Exception();
    }
}
catch (ArithmeticException e) {
    System.out.println("Kerzen");
}
catch (Exception e) {
    System.out.println("");
}
}
```

## Musterlösungen zu den Selbsttestaufgaben

### Aufgabe 1: Der intelligente Kühlschrank

Die Kühlschrankwelt kann z.B. die folgenden Gegenstände enthalten:

1. den Kühlschrankbesitzer
2. den Kühlschrank
3. den Hausroboter
4. den Reparaturdienst
5. die Lieferanten: Lebensmittelhändler, Optiker, Weinhändler
6. die Bank

Zu diesen Gegenständen gehören die folgenden Methoden:

1. Methoden des Kühlschranks:
  - Testen-auf-vorhandenen-Strom
  - Testen-auf-korrekte-Temperatur
  - Erhöhen-der-Temperatur
  - Verringern-der-Temperatur
  - Senden-einer-Nachricht-an-Kühlschrankbesitzer
  - Entnehmen-von-Kühlschrankinhalten
  - Hinzufügen-von-Kühlschrankinhalten
  - Überprüfen-des-Kühlschrankinhalts
  - Speichern-des-Kühlschrankinhalts
  - Nahrungsmittelbestellung-bei-Bedarf
  - Kontaktlinsenmittelbestellung-bei-Bedarf
  - Champagner-und-Weissweinbestellung-bei-Bedarf
  - Bezahlen-der-Lieferung
2. Methoden des Roboters:
  - Reagieren-auf-das-Türklingeln
  - Türöffnen
  - Lieferung-entgegennehmen
  - Beladen-des-Kühlschranks



## 3. Methoden des Reparaturdienst

- Reparaturwunsch-entgegennehmen
- Reparatur-durchfuehren
- Rechnung-senden

## 4. Methoden des Nahrungsmittelhaendlers

- Lieferwunsch-entgegennehmen
- Lieferung-ausfuehren
- Rechnung-schreiben

## 5. Methoden des Optikers

- Lieferwunsch-entgegennehmen
- Lieferung-ausfuehren
- Rechnung-schreiben

## 6. Methoden des Weinhaendlers

- Lieferwunsch-entgegennehmen
- Lieferung-ausfuehren
- Rechnung-schreiben

## 7. Methoden der Bank

- Kundenkonto-fuehren
- Ueberweisungsauftrag-annehmen

**Aufgabe 2: Erste Schritte in Java**

Zu dieser Aufgabe gibt es keine Musterlösung.

**Aufgabe 3: Implizite Typkonvertierungen**

```
class Main {  
    public static void main( String[] args ) {  
(1)  int i = 10;  
(2)  long l = 55567843L;  
(3)  byte by = 15;  
(4)  boolean b = true;  
(5)  double d = 1.25;  
(6)  l = i;  
(7)  d = l;  
    }
```

```
(8)  by = i;  
(9)  l = l + by;  
(10) by = by - b;  
(11) d = (l / i) * 20;  
    }  
}
```

**Implizite Typkonvertierungen** Folgende erweiternden impliziten Typkonvertierungen werden für primitive Datentypen vorgenommen:

`byte` → `short` → `int` → `long` → `float` → `double` und `char` → `int`

- In Zeile (6) wird eine implizite Typkonvertierung des in `i` enthaltenen `int`-Wertes in einen `long`-Wert vorgenommen.
- In Zeile (7) wird eine implizite Typkonvertierung des in `l` enthaltenen `long`-Wertes in einen `double`-Wert vorgenommen.
- In Zeile (9) wird vor Ausführung der Addition eine implizite Typkonvertierung des in `by` enthaltenen `byte`-Wertes in einen `long`-Wert vorgenommen.
- Um die Division in Zeile (11) durchführen zu können, wird zunächst der in `i` enthaltene `int`-Wert in einen `long`-Wert konvertiert. Das Ergebnis der Division ist vom Typ `long`. Um die anschließende Multiplikation mit 20 durchführen zu können, wird der Wert 20 in einen `long`-Wert konvertiert. Bei der Zuweisung des Ergebnisses an die Variable `d` wird das Ergebnis in einen Wert vom Typ `double` konvertiert.

**Ungültige Zuweisungen / Operationen** Obiges Programm enthält folgende Fehler:

- In Zeile (8) müsste ein `int`-Wert einem `byte`-Wert zugewiesen werden. Da `int`-Werte größer sein können als der größte `byte`-Wert, könnte bei der Zuweisung ein Genauigkeitsverlust auftreten. Der Compiler weist diese Zuweisung daher zurück.
- Werte vom Typ `byte` sind inkompatibel mit Werten vom Typ `boolean`. Daher ist die Subtraktion in Zeile (10) unzulässig.

#### Aufgabe 4: Schleifen

Folgendes einfache Java-Programm verwendet zunächst die `while`-Schleife zur Ausgabe aller beim Programmaufruf mitgegebenen Programmparameter, dann die `do`-Schleife und anschließend beide Varianten der `for`-Schleife. Bei der `do`-Schleife muss darauf geachtet werden, dass die Ausführung des Schleifenrumpfes verhindert wird, wenn überhaupt keine Argumente beim

Programmaufruf mitgegeben werden. Dies wird durch Voranstellen der `if`-Anweisung erreicht.

```
public class Schleifen {
    public static void main(String[] args) {
        //Mit while-Schleife
        int i = 0;
        while (i < args.length) {
            System.out.println(args[i]);
            i = i + 1;
        }
        //Mit do-Schleife
        i = 0;
        if (args.length > 0)
            do {
                System.out.println(args[i]);
                i = i + 1;
            } while (i < args.length);
        // Mit erster Variante der for-Schleife
        for (i = 0; i < args.length; i = i + 1)
            System.out.println(args[i]);
        // Mit zweiter Variante der for-Schleife
        for (String arg : args)
            System.out.println(arg);
    }
}
```

### Aufgabe 5: Kontrollstrukturen

Folgendes Java-Programm löst die gestellte Aufgabe:

```
public class Ggt {
    public static void main(String[] args) {
        if (args.length <= 1) return;
        int z1, z2, ggt;
        z1 = Integer.parseInt(args[0]);
        z2 = Integer.parseInt(args[1]);

        // Mit der kleinsten uebergebenen Zahl als
        // Teiler starten.
        if (z1 <= z2) ggt = z1;
        else
            ggt = z2;

        // Solange ggt nicht beide Zahlen z1 und z2 teilt,
        // verringere ggt um eins. Die Schleife bricht
```

```

// spaetestens bei ggt == 1 ab.
while ( (z1 % ggt) != 0 || (z2 % ggt) != 0) {
    ggt = ggt - 1;
}
switch (ggt) {
    case 1: {
        System.out.println
            (" Der groesste gemeinsame Teiler ist 1.");
        break;
    }
    case 2: {
        System.out.println
            (" Der groesste gemeinsame Teiler ist 2.");
        break;
    }
    case 3: {
        System.out.println
            (" Der groesste gemeinsame Teiler ist 3.");
        break;
    }
    case 4: {
        System.out.println
            (" Der groesste gemeinsame Teiler ist 4.");
        break;
    }
    default:
        System.out.println
            (" Der groesste gemeinsame Teiler ist groesser als 4.");
}
}
}

```

## Aufgabe 6: Sortieren eines Feldes

### Teilaufgabe 1: Eliminieren von Syntaxfehlern

Die Syntaxfehler sind im Folgenden in #, die Fehlerbeschreibung in eine Folge von 3 Sternen eingeschlossen.

```

class Sortieren {
    public static void main (#s#tring[] args) {
        *** Der vordefinierte Datentyp fuer Zeichenketten ***
        *** ist String, nicht string. ***

        // Ein double Feld erzeugen, das genauso gross ist
        // wie das args-Feld
    }
}

```

```

double[] field = new double[args.length] ;
    *** Der vordefinierte Datentyp fuer double-Werte      ***
    *** ist double, nicht double.                          ***
    *** Bei der Erzeugung eines Arrays wird dessen         ***
    *** Groesse in eckige Klammern [...] eingeschlossen.  ***
    *** Jede Anweisung muss mit einem Semikolon abge-    ***
    *** schlossen werden.                                  ***

// Alle Zahlen, die in args als Strings vorliegen
// in double-Werte umwandeln und in das Feld field eintragen
for (int i = 0; i < args.length; i = i + 1) {
    *** Eine for-Schleife beginnt mit dem Schlueselwort   ***
    *** for und nicht For. Zur Initialisierung einer      ***
    *** Laufvariablen der for-Schleife wird eine         ***
    *** Zuweisung (=) verwendet, kein Vergleich (==).    ***
    *** Die Angaben fuer die Laufvariable muessen in     ***
    *** runde statt in geschweifte Klammern eingeschlossen ***
    *** werden. Die einzelnen Anweisungen innerhalb der  ***
    *** Klammern muessen durch ein Semikolon abgeschlossen ***
    *** werden, nicht durch ein Komma.                    ***

    field[i] = Double.parseDouble(args[i]);
    *** Gross- und Kleinschreibung ist in Java relevant.  ***
    *** Die verwendete Variable fuer das Array, das die   ***
    *** double-Werte aufnehmen soll, ist weiter oben mit  ***
    *** dem Namen field deklariert worden. Feld waere eine ***
    *** von field verschiedene Variable.                  ***
    *** Ein Element eines Arrays wird ueber seinen Index  ***
    *** angesprochen, der in eckige Klammern gesetzt     ***
    *** werden muss. Runde Klammern sind falsch.          ***
}

// Hier Programmcode zum Sortieren einfuegen

// Hier Programmcode zur Bestimmung und
// Ausgabe des groessten Elements einfuegen

// Den Inhalt des Feldes field ausgeben
for (int i = 0; i < args.length; i = i + 1) {
    *** Eine Zuweisung erfolgt durch =, nicht durch      ***
    *** := wie in PASCAL.                                  ***

    System.out.println (i + ". " + field[i]);
    *** Aktuelle Parameter werden in runde Klammern      ***
    *** eingeschlossen.                                    ***

}

*** Statt der eckigen Klammer muss eine geschweifte     ***
*** Klammer stehen.                                       ***
}

```

Das syntaktisch korrekte Programmfragment sieht wie folgt aus:

```
class Sortieren {
    public static void main(String[] args) {
        // Ein double Feld erzeugen, das genauso gross ist
        // wie das args-Feld.
        double[] feld = new double[args.length];

        // Alle Zahlen, die in args als Strings vorliegen,
        // in double-Werte umwandeln und in das Feld feld eintragen.
        for (int i = 0; i < args.length; i = i + 1) {
            feld[i] = Double.parseDouble(args[i]);
        }

        // Hier Programmcode zum Sortieren einfuegen

        // Hier Programmcode zur Bestimmung und
        // Ausgabe des groessten Elements einfuegen

        // Den Inhalt des Feldes feld ausgeben
        for (int i = 0; i < args.length; i = i + 1) {
            System.out.println(i + ". " + feld[i]);
        }
    }
}
```

### Teilaufgabe 2: Erweiterung des korrigierten Programms

Da das Feld absteigend sortiert wird, muss das größte Element das erste Element des Feldes sein und muss nicht erst durch Suchen ermittelt werden.

```
class Sortieren {
    public static void main(String[] args) {
        // Ein double Feld erzeugen, das genauso gross
        // ist wie das args-Feld
        double[] feld = new double[args.length];

        // Alle Zahlen, die in args als Strings vorliegen
        // in double-Werte umwandeln und in das Feld feld eintragen
        for (int i = 0; i < args.length; i = i + 1) {
            feld[i] = Double.parseDouble(args[i]);
        }

        // Feld absteigend sortieren
        for (int i = 0; i < args.length - 1; i = i + 1) {
            int max = i;
            for (int j = i + 1; j < args.length; j = j + 1) {
                if (feld[j] > feld[max]) {
```

```

        max = j;
    }
}
double h = feld[i];
feld[i] = feld[max];
feld[max] = h;
}

// Ausgabe des groessten uebergebenen Wertes.
// Wir gehen bei dieser Loesung davon aus, dass
// mindestens ein Programmparameter uebergeben wird.
System.out.println("Das groesste Element ist " + feld[0]);

// Den Inhalt des Feldes feld ausgeben
for (int i = 0; i < args.length; i = i + 1) {
    System.out.println(i + ". " + feld[i]);
}
}
}

```

### Aufgabe 7: Ausnahmebehandlung (Brandschutzübung)

1. Das Motto der diesjährigen Brandschutzübung lautet:

```

man
spielt
nicht
mit
Feuer

```

2. Um dieses Ergebnis zu finden, muss man Schritt für Schritt die Ausführung des Programms nachvollziehen:
  1.  $7\%5$  ergibt 2 und  $i\%2$  somit 0, d.h. es wird eine `ArithmeticException` wegen der Division durch null bei  $((i/(i\%2)))$  geworfen.
  2. Diese wird mit `catch(Exception e)` abgefangen und es wird man ausgegeben.
  3. Da  $((7\%6/(7\%6\%2))\neq 1)$  wahr ist, wird eine Ausnahme vom Typ `Exception` erzeugt und bei `catch(Exception u)` abgefangen. Dann wird `spielt` ausgegeben.
  4. `nicht` wird ausgegeben.
  5.  $true\&false?0:1$  ergibt 1, somit wird nach `case 1: mit` ausgegeben.

6. Da der `case 1:` Fall keine `break`-Anweisung enthält, wird noch die Anweisung des `default`-Falles ausgeführt und mit `throw new Exception()` eine neue Ausnahme vom Typ `Exception` erzeugt.
7. Diese wird mit `catch(Exception e)` abgefangen und es wird `Feuer` ausgegeben.
8. Nun wird der `finally`-Teil ausgeführt. `false && true ? 0 : 2` ergibt 2 und im `default`-Fall wird eine Ausnahme vom Typ `Exception` geworfen. Diese wird wiederum mit `catch(Exception e)` abgefangen, jedoch wird nichts mehr ausgegeben.





# Studierhinweise zur Kurseinheit 2

Diese Kurseinheit beschäftigt sich mit dem zweiten Kapitel des Kurstextes. Sie sollten dieses Kapitel im Detail studieren und verstehen. Nehmen Sie sich Zeit, die Sprachkonstrukte an kleinen, selbst entworfenen Beispielen im Rahmen dieser Kurseinheit zu üben! Bearbeiten Sie auch die Selbsttestaufgaben am Ende der Kurseinheit. Wie bereits bei der ersten Kurseinheit gesagt, reicht ein bloßes Durchlesen des Textes nicht aus.

Prüfen Sie deshalb nach dem Durcharbeiten des Kurstextes, ob Sie die Lernziele erreicht haben. Stellen Sie sich dazu vor, dass die Lernziele als Fragen formuliert sind, und versuchen Sie, diese zu beantworten.

## **Lernziele:**

- Grundbegriffe objektorientierter Sprachen: Objekt, Klasse, Methode, Attribut.
- Entwerfen von Klassen.
- Wichtige Sprachkonstrukte, die über einen objektorientierten Sprachkern hinausgehen, am Beispiel von Java.
- Entwicklung und Benutzung rekursiver Klassen.
- Parametrisierung von Klassen.
- Verständnis für das Trennen von Schnittstelle und Implementierung und für die Ziele der Kapselung von Implementierungsteilen.
- Konstrukte zum Strukturieren von Klassen: Schachtelung und Pakete.



# Kapitel 2

## Objekte, Klassen, Kapselung

Dieses Kapitel erläutert, wie Objekte beschrieben werden können, und geht dazu ausführlich auf das Klassenkonzept ein. Es behandelt Kapselungsaspekte und stellt Techniken vor, mit denen sich Mengen von Klassen strukturieren und modularisieren lassen. Zu allen diesen Aspekten werden die entsprechenden Sprachkonstrukte von Java vorgestellt. Um die konzeptionelle Trennung zwischen Klassenkonzept einerseits und Vererbung andererseits deutlich zu machen, werden Subtyping und Vererbung erst im nächsten Kapitel behandelt (siehe Kap. 3).

Dieses Kapitel besteht aus zwei Abschnitten. Im Mittelpunkt des ersten Abschnitts steht die Deklaration und Verwendung von Klassen. Dabei wird auch auf den Entwurf von Klassen eingegangen, und es werden weitere in diesem Zusammenhang wichtige Aspekte behandelt, insbesondere:

- rekursive Klassendefinitionen,
- parametrische Klassen, d.h. Klassen mit Typparametern,
- die Verwendung von Klasseninformation zur Programmlaufzeit.

Der zweite Abschnitt erläutert Techniken und Konstrukte zur Kapselung von Implementierungsteilen und geht auf die Schachtelung von Klassen und die Zusammenfassung von Klassen in Pakete ein.

### 2.1 Objekte und Klassen

Dieser Abschnitt behandelt die Frage: Was sind Objekte und wie werden sie in Programmen beschrieben? Er geht dazu vom Grundmodell der objektorientierten Programmierung aus (vgl. die Abschnitte 1.1.1 und 1.2.3) und stellt Sprachkonzepte zur Beschreibung von Objekten vor. Sein Hauptteil widmet sich dem Klassenkonzept von Java und dessen sprachlichen Erweiterungen.

Programmiersprachen, die es ermöglichen, Objekte zu definieren, zu erzeugen und ihnen Nachrichten zu schicken, die aber weder Vererbung noch

objektbasiert

Subtyping unterstützen, werden vielfach *objektbasierte* Sprachen genannt. Dieser Abschnitt beschäftigt sich also insbesondere mit dem objektbasierten Kern der Sprache Java.

### 2.1.1 Beschreibung von Objekten

Die objektorientierte Programmierung betrachtet eine Programmausführung als ein System kooperierender Objekte. Ein objektorientiertes Programm muss also im Wesentlichen beschreiben, wie die Objekte aussehen, die während der Programmausführung existieren, wie sie erzeugt bzw. gelöscht werden und wie die Kommunikation zwischen den Objekten angestoßen wird. Grundsätzlich gibt es zwei Konzepte zur programmiersprachlichen Beschreibung von Objekten:

Klassenkonzept

1. *Klassenkonzept*: Der Programmierer beschreibt nicht einzelne Objekte, sondern deklariert Klassen. Eine Klasse ist eine *Beschreibung* der Eigenschaften, die die Objekte dieser Klasse haben sollen. Die Programmiersprache ermöglicht es dann, während der Programmausführung Objekte zu deklarierten Klassen zu erzeugen. Die Klassen können zur Ausführungszeit nicht verändert werden. Klassendeklarationen entsprechen damit der Deklaration von Verbundtypen imperativer Sprachen, bieten aber zusätzliche Möglichkeiten.

Prototyp-  
Konzept  
Klonen

2. *Prototyp-Konzept*: Der Programmierer beschreibt direkt einzelne Objekte. Neue Objekte werden durch Klonen existierender Objekte und Verändern ihrer Eigenschaften zur Ausführungszeit erzeugt. *Klonen* eines Objekts *obj* meint dabei das Erzeugen eines neuen Objekts, das die gleichen Eigenschaften wie *obj* besitzt. Verändern bedeutet zum Beispiel, dass dem Objekt weitere Attribute hinzugefügt werden oder dass Methoden durch andere ersetzt werden.

Das Klassenkonzept ermöglicht eine bessere statische Überprüfung von Programmen, da zur Übersetzungszeit alle wesentlichen Eigenschaften der Objekte bekannt sind. Insbesondere kann Typkorrektheit überprüft werden und damit einhergehend, ob jedes Objekt, dem eine Nachricht *m* geschickt wird, auch eine Methode besitzt, um *m* zu bearbeiten. Beim Prototyp-Konzept können diese Prüfungen im Allgemeinen nur dynamisch durchgeführt werden, da die Eigenschaften der Objekte erst zur Ausführungszeit festgelegt werden. Dafür ist das Prototyp-Konzept flexibler (vgl. [Bla94] bzgl. der Programmierung mit Prototypen). Beim Entwurf einer Programmiersprache müssen diese Vor- und Nachteile abgewogen werden.

Es sind auch Mischformen zwischen den beiden Konzepten möglich. Zunächst behandeln wir allerdings das Klassenkonzept und seine Aus-

prägung in Java. In Kap. 3 bzw. 4 werden wir aber auch das Erzeugen einzelner Objekte und das Klonen von Objekten kennen lernen<sup>1</sup>.

Aus programmiersprachlicher Sicht ergeben sich die *Eigenschaften* und das *Verhalten* eines Objekts aus dessen möglichen Zuständen und daraus, wie es auf Nachrichten reagiert. Demzufolge muss die Beschreibung von Objekten – also insbesondere eine Klassendeklaration – festlegen,

*Eigenschaft**Verhalten*

- welche Zustände ein Objekt annehmen kann,
- auf welche Nachrichten es reagieren soll und kann
- und wie die Methoden aussehen, mit denen ein Objekt auf den Empfang von Nachrichten reagiert.

Die Menge der Zustände eines Objekts wird durch die Attribute beschrieben, die es besitzt. Ein Attribut ist eine objektlokale Variable (vgl. Abschn. 1.2.3). Der aktuelle *Zustand* eines Objekts *obj* ist bestimmt durch die aktuellen Werte seiner Attribute. Die Menge der Zustände entspricht dann allen möglichen Wertkombinationen, die die Attribute von *obj* annehmen können. Üblicherweise reagiert ein Objekt genau auf die Nachrichten, für die es Methoden besitzt. Außer den Attributen müssen also nur noch die Methoden beschrieben werden. Der folgende Abschnitt erläutert, wie Attribute und Methoden in Java deklariert werden, wie Objekte erzeugt werden und wie auf Attribute zugegriffen werden kann.

*aktueller  
Zustand*

## 2.1.2 Klassen beschreiben Objekte

Eine *Klasse* beschreibt in Java einen neuen Typ von Objekten und liefert eine Implementierung für diese Objekte. Der Rumpf der Klasse besteht normalerweise aus einer Liste von Attribut-, Methoden- und Konstruktordeklarationen. Eine Attributdeklaration sieht genauso aus wie eine Variablendeklaration (vgl. Abschn. 1.3.1.2, S. 28); sie legt den Typ und den Namen des Attributs fest. Eine Methodendeklaration legt den Namen der Methode fest und beschreibt deren formale Parameter und mögliche Ergebnisse. Konstruktoren werden in Java benutzt, um Objekte zu initialisieren. Konstruktordeklarationen ähneln Methodendeklarationen, besitzen aber keinen Rückgabotyp und haben immer den Namen der umfassenden Klasse. Wir betrachten zunächst ein Beispiel und erläutern dann die Sprachkonstrukte im Einzelnen.

*Klasse*

Als Beispiel betrachten wir eine Klasse, die Textseiten mit Titel und Inhalt beschreibt und uns als Ausgangspunkt für eine Browseranwendung dienen wird (vgl. Abb. 2.4, S. 86). Da wir dieses Beispiel im Folgenden ausbauen werden, um in einige Aspekte des World Wide Web, kurz WWW, einzuführen, nennen wir die Klasse `W3Seite`:

<sup>1</sup>Das Verändern der Eigenschaften von Objekten zur Ausführungszeit ist in Java nicht möglich.

```
(1)  class W3Seite {
(2)      String titel;                // Attribut
(3)      String inhalt;              // Attribut

(4)      W3Seite ( String t, String i ) {    // Konstruktor
(5)          this.titel  = t;
(6)          this.inhalt = i;
(7)      }

(8)      String getTitel() {                // Methode
(9)          return this.titel;
(10)     }
(11)     String getInhalt() {                // Methode
(12)         return this.inhalt;
(13)     }
(14) }
```

Die Klasse `W3Seite` deklariert die beiden Attribute `titel` und `inhalt`, beide vom Typ `String`, einen Konstruktor mit zwei Parametern zum Erzeugen und Initialisieren von Objekten der Klasse `W3Seite` sowie die beiden Methoden `getTitel` und `getInhalt` zum Auslesen der Attributwerte. (Die genaue Bedeutung von `this` wird auf Seite 79 erläutert. Hier ist zunächst nur wichtig, dass `this` dazu verwendet wird, die Attribute der Klasse eindeutig zu identifizieren<sup>2</sup>.) Eine triviale Anwendung der Klasse demonstrieren wir mit folgender Klasse `W3SeitenTest`, die die Klasse `W3Seite` zu einem ausführbaren Programm ergänzt. (Auf die Bedeutung der Klammern in `(System.out).println (...)` in Zeile (21) wird auf Seite 81 näher eingegangen.)

```
(15) class W3SeitenTest {
(16)     public static void main( String[] args ) {
(17)         W3Seite  meineSeite;
(18)         meineSeite =
(19)             new W3Seite( "Abstraktion",
(20)                         "Abstraktion bedeutet ..." );
(21)         (System.out).println( meineSeite.getInhalt() );

(22)         W3Seite  meinAlias;
(23)         meinAlias = meineSeite;
(24)         meinAlias.inhalt = "Keine Angabe";
(25)         System.out.println( meineSeite.getInhalt() );
(26)     }
(27) }
```

---

<sup>2</sup>Dies gilt nur für nicht statische Attribute, wie wir später noch sehen werden. Allerdings sind nicht statische Attribute der Normalfall.

In Zeile (17) wird die *lokale* Variable `meineSeite` vom Typ `W3Seite` deklariert. In Zeile (18)-(20) wird ihr (die Referenz auf) ein neues Objekt vom Typ `W3Seite` zugewiesen. Der Konstruktor initialisiert dabei das Attribut `titel` zu "Abstraktion" und das Attribut `inhalt` zu "Abstraktion bedeutet ...". In Zeile (21) wird für das Objekt, das von der Variablen `meineSeite` referenziert wird, die Methode `getInhalt` aufgerufen. Sie liefert die Zeichenreihe "Abstraktion bedeutet ..." als aktuellen Parameter für die Methode `println`.

Die Zeilen (1)-(21) enthalten alle wesentlichen syntaktischen Konstrukte für die objektbasierte Programmierung in Java. Die letzten vier Zeilen sollen auf einen wichtigen Aspekt in der objektbasierten Programmierung hinweisen und den Unterschied zwischen Variablen, Objekten und Referenzen auf Objekte nochmals verdeutlichen. In Zeile (22) wird die lokale Variable `meinAlias` deklariert; in Zeile (23) wird ihr (die Referenz auf) dasselbe Objekt zugewiesen, das auch von der Variablen `meineSeite` referenziert wird. Damit existieren zwei Variablen, die dasselbe Objekt referenzieren. Wird ein Objekt von zwei oder mehr Variablen referenziert, spricht man häufig von *Aliasing* und nennt Variablen wie `meinAlias` auch einen *Alias* für das schon referenzierte Objekt. In Zeile (24) wird dem Attribut `inhalt` des von `meinAlias` referenzierten Objekts die Zeichenreihe "Keine Angabe" zugewiesen. Die daraus resultierende Variablenbelegung ist in Abb. 2.1 dargestellt (eine Erklärung der Repräsentation von String-Objekten verschieben wir bis Abschn. 2.1.4). In Zeile (25) wird demzufolge die Zeichenreihe "Keine Angabe" ausgegeben.

*Aliasing*

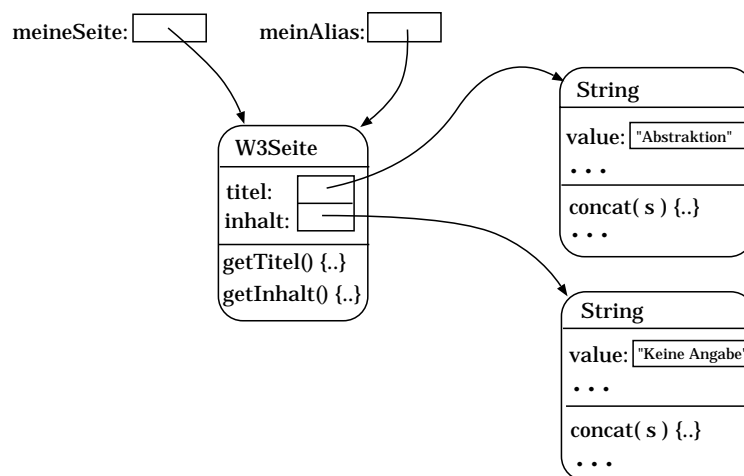


Abbildung 2.1: Variablen- und Attributbelegung zum Beispiel

Die folgenden Paragraphen liefern eine genauere und allgemeinere Erläuterung der vom obigen Beispiel illustrierten Sprachkonstrukte.



Klassen-  
deklaration

**Klassendeklarationen, Attribute und Objekte.** Betrachtet man kein Subtyping und keine Vererbung, besteht eine Klassendeklaration in Java aus einer Modifikatorenliste, dem Schlüsselwort `class`, dem Klassennamen und einer in geschweifte Klammern eingeschlossenen Liste von Attribut-, Konstruktor- und Methodendeklarationen:

```
Modifikatorenliste class Klassenname {
    Liste von Attribut-, Konstruktor- und Methodendeklarationen
}
```

Klassentyp

Die Modifikatorenliste ist eine Liste von Schlüsselwörtern, mit denen bestimmte Eigenschaften der Klasse beschrieben werden können; welche Eigenschaften das sind, werden wir im Laufe des Kurses kennen lernen. Sie kann wie in den obigen Beispielen leer sein. Der Klassenname wird gleichzeitig als Typname für die Objekte dieser Klasse verwendet<sup>3</sup>. Typen, die durch eine Klasse deklariert sind, nennen wir *Klassentypen*. Die Attribute, Konstruktoren und Methoden einer Klasse nennen wir zusammenfassend die *Komponenten* der Klasse.

objektlokale  
Variable

Ein Objekt einer Klasse besitzt für jedes Attribut der Klasse eine *objektlokale Variable*. Der Einfachheit halber werden wir diese objektlokalen Variablen ebenfalls als *Attribute* bezeichnen. In der objektorientierten Literatur wird allerdings zum Teil in der Begriffswahl zwischen der Klassenebene und der Objektebene unterschieden. Die Objekte einer Klasse werden dann häufig die *Instanzen* der Klasse genannt, und man spricht statt vom Erzeugen eines Objekts vom *Instanziiieren* einer Klasse. Dementsprechend bezeichnet man die objektlokalen Variablen vielfach auch als *Instanzvariablen*. (Bei dem Wort „Instanz“ sollte man nicht an behördliche Instanzen denken, sondern vielmehr an das Ausstanzen eines Gegenstands gemäß einer vorgegebenen Schablone: die Klasse entspricht der Schablone, das Objekt dem Gegenstand.<sup>4</sup>)

Instanz

Instanziiieren

Instanzvariable

**Methodendeklaration.** In Java besteht eine Methodendeklaration aus der *erweiterten Methodensignatur* und dem *Methodenrumpf*. Der Methodenrumpf ist ein Anweisungsblock (zur Definition von Anweisungsblöcken siehe Unterabschn. 1.3.1.3); die erweiterte Methodensignatur hat im Allgemeinen folgende Form:

```
Modifikatorenliste Ergebnistyp Methodenname (Parameterliste) throws-Deklaration
```

Wir gehen die Teile der erweiterten Methodensignatur kurz durch:

- Wie bei Klassen, ist die Modifikatorenliste eine Liste von Schlüsselwörtern, mit denen bestimmte Eigenschaften der Methoden beschrieben

<sup>3</sup>In Kap. 3 werden wir sehen, dass der Typ, der von einer Klasse deklariert wird, im Allgemeinen nicht nur die Objekte dieser Klasse umfasst.

<sup>4</sup>Um die angedeutete falsche Analogie zu vermeiden, wird in [Goo99] der Terminus „Ausprägung“ statt „Instanz“ verwendet.

werden können. Sie kann leer sein wie bei den Methoden `getTitle` und `getInhalt`. Ein Beispiel für eine nichtleere Modifikatorenliste bietet die Methodendeklaration von `main`, wie wir sie in unseren Testprogrammen verwenden (vgl. Zeile (16) auf S. 76): Sie enthält die Modifikatoren `public` und `static`, deren Bedeutung wir noch in diesem Kapitel behandeln werden.

- Als Ergebnistyp ist entweder ein Typname oder, wenn die Methode kein Ergebnis zurückliefert, das Schlüsselwort `void` zulässig. Die Methoden aus obigem Beispiel liefern also (Referenzen auf) String-Objekte zurück; die Methode `main`, die den Programmeinstiegspunkt festlegt, liefert kein Ergebnis.
- Nach dem Methodennamen folgt die in Klammern eingeschlossene und durch Komma getrennte Liste von Parameterdeklarationen. Eine Parameterdeklaration besteht aus dem Typ und dem Namen des Parameters; ein Beispiel zeigt die Deklaration von `main` in Zeile (16), die einen Parameter `args` vom Typ `String[]` deklariert. Die Methodendeklarationen `getTitle` und `getInhalt` illustrieren, dass Parameterlisten auch leer sein können.
- In der *throws-Deklaration* muss deklariert werden, welche Ausnahmen, die in der Methode selbst *nicht* behandelt werden, bei ihrer Ausführung auftreten können (vgl. die Erläuterungen zur `try`-Anweisung auf den Seiten 38ff); können keine Ausnahmen auftreten, kann die *throws-Deklaration* entfallen (wie bei den obigen Methodendeklarationen). So wie der Ergebnistyp darüber Auskunft gibt, welche Ergebnisse die Methode bei normaler Terminierung liefert, gibt die *throws-Deklaration* darüber Auskunft, welche Ausnahmen bei abrupter Terminierung der Methode möglicherweise auftreten. Details zur *throws-Deklaration* werden in Kap. 4 behandelt.

Im Rumpf einer Methode *m* sind die Parameter von *m* sowie alle Attribute und Methoden der umfassenden Klasse sichtbar, also auch diejenigen, die erst weiter unten im Text der Klassendeklaration stehen. Wie bereits in Abschnitt 1.3.2.1 erwähnt, besitzt jede Methode zusätzlich zu den explizit deklarierten formalen Parametern einen weiteren, impliziten Parameter, der bei einem Aufruf der Methode eine Referenz auf das Objekt aufnimmt, auf dem die Methode aufgerufen wurde. Häufig nennt man dieses Objekt auch einfach das *this-Objekt*, oder *self-Objekt*<sup>5</sup>. Daher hat dieser implizite Parameter den Namen `this` und kann unter dieser Bezeichnung im Rumpf der Methode verwendet werden (vgl. Zeile (9) des Beispiels und Abschn. 1.3.2.1).

*impliziter  
Parameter*

<sup>5</sup>Beispielsweise wird es in Java und C++ als *this-Objekt* bezeichnet, in Smalltalk als *self-Objekt*; die Bezeichnung *impliziter Parameter* rührt daher, dass der Parameter in der Methodendeklaration nicht explizit erwähnt wird.

**Konstruktordeklaration.** Eine Konstruktordeklaration besteht in Java aus der *erweiterten Konstruktorsignatur* und dem *Konstruktorrumpf*. Die erweiterte Konstruktorsignatur hat im Allgemeinen folgende Form:

*Modifikatorenliste* *Klassenname* (*Parameterliste*) *throws-Deklaration*

Der Klassenname muss gleich dem Namen der umfassenden Klasse sein. Er wird häufig auch als Name des Konstruktors verwendet. Der Konstruktorrumpf ist ein Anweisungsblock (zur Definition von Anweisungsblöcken siehe Unterabschn. 1.3.1.3). Er darf `return`-Anweisungen ohne Ergebnis und als erste Anweisung eine spezielle Form von Konstruktoraufrufen enthalten (darauf werden wir in Abschn. 3.3 näher eingehen). Im Rumpf eines Konstruktors sind seine Parameter sowie alle Attribute und Methoden der umfassenden Klasse sichtbar.

Konstrukturen dienen zum Initialisieren von Objekten (vgl. die Deklaration des Konstruktors von `W3Seite`, Zeilen (4)–(7), und dessen Aufruf in den Zeilen (19)–(20)). In Konstruktorrümpfen bezeichnet `this` das zu initialisierende Objekt. Enthält eine Klassendeklaration keinen Konstruktor, stellt Java automatisch einen Konstruktor mit leerer Parameterliste und leerem Rumpf bereit; dieser implizite Konstruktor wird *default-Konstruktor* genannt. Weitere wichtige Eigenschaften von Konstruktoren werden in Abschn. 3.3 behandelt.

default-  
Konstruktor

**Objekterzeugung, Attributzugriff und Methodenaufruf.** Die obigen beiden Absätze haben die Deklaration von Klassen, Methoden und Konstruktoren behandelt. Dieser Absatz erläutert, wie deren Anwendung in Programmen beschrieben wird; d.h. wie Objekte erzeugt werden, wie auf deren Attribute zugegriffen wird und wie man deren Methoden aufruft. Dazu erweitern wir die in Abschn. 1.3.1.2 vorgestellte Ausdruckssyntax um drei zusätzliche Ausdrucksformen: die Objekterzeugung, den Attributzugriff und den Methodenaufruf.

Syntaktisch ist eine *Objekterzeugung* ein Ausdruck, der mit dem Schlüsselwort `new` eingeleitet wird und folgenden Aufbau hat, wobei die aktuelle Parameterliste eine Liste von Ausdrücken ist:

`new` *Klassenname* ( *AktuelleParameterliste* )

Die Objekterzeugung wird wie folgt ausgewertet: Zuerst wird eine neue Instanz der Klasse *Klassenname* erzeugt. Die Attribute dieses neu erzeugten Objekts sind mit default-Werten initialisiert (der default-Wert für ein Attribut von einem Klassentyp ist die Referenz `null`, der default-Wert von zahlenwertigen Attributen ist 0, der für boolesche Attribute `false`). Danach werden die Ausdrücke der Parameterliste ausgewertet. Anschließend wird der Konstruktor *Klassenname* aufgerufen. Bei diesem *Konstruktoraufruf* wird das neu erzeugte Objekt als impliziter Parameter übergeben. Terminiert die Auswertung einer Objekterzeugung normal, liefert sie die Referenz auf ein neu erzeugtes Objekt der Klasse *Klassenname* als Ergebnis. Abrupte Terminierung

tritt auf, wenn nicht mehr genügend Speicher für die Allokation des Objekts in der Laufzeitumgebung zur Verfügung steht oder wenn die Auswertung der Parameterausdrücke oder des Konstruktoraufrufs abrupt terminiert. Die Zeilen (19) und (20) des obigen Beispiels demonstrieren eine Objekterzeugung.

Ein *Attributzugriff* hat grundsätzlich die Form *Ausdruck* . *Attributname*. Der Typ des Ausdrucks muss ein Klassentyp sein und die zugehörige Klasse muss ein Attribut des angegebenen Namens besitzen. Liefert die Auswertung des Ausdrucks den Wert `null`, terminiert der Attributzugriff abrupt und erzeugt eine `NullPointerException`. Im Normalfall liefert der Ausdruck ein Objekt *X* vom angesprochenen Klassentyp.

Steht der Attributzugriff auf der linken Seite einer Zuweisung, wird das Attribut *Attributname* vom Objekt *X* durch die Zuweisung entsprechend modifiziert (schreibender Zugriff). Andernfalls wird der Wert des Attributs gelesen. Die Zeilen (5), (6), (9), (12) und (24) des Beispiels zeigen Attributzugriffe in Standardform. Der Typ der Variablen `meinAlias` und des Vorkommens von `this` in Zeile (9) ist `W3Seite`. In der Zeilen (9) und (12) wird der Wert des Attributs gelesen, die Zeilen (5), (6) und (24) illustrieren einen schreibenden Zugriff.

Als abkürzende Schreibweise gestattet es Java statt `this.Attributname` einfach nur den Attributnamen zu schreiben, sofern keine Namenskonflikte auftreten. In Zeile (5) könnten wir statt `this.titel = t;` daher auch einfach `titel = t;` schreiben. Zum präzisen Verständnis der Bedeutung eines Programmes ist es wichtig, sich dieser abkürzenden Schreibweise immer bewusst zu sein. Insbesondere kann der Zugriff auf eine lokale Variable einer Methode syntaktisch leicht mit einem Attributzugriff verwechselt werden. Hieße der erste Parameter des Konstruktors von `W3Seite` `titel` statt `t`, müßte die Zuweisung `titel = t;` geändert werden in `titel = titel;`. Hier wäre aber nicht mehr klar, wo das Attribut und wo der Parameter gemeint ist. Zur Auflösung der Namenskollision muss bei der Zuweisung in diesem Fall der `this`-Parameter zur Identifikation verwendet werden: `this.titel = titel;`

Ein *Methodenaufruf* ähnelt syntaktisch einem Attributzugriff:

*Ausdruck* . *Methodenname* ( *AktuelleParameterliste* )

Der Ausdruck beim Methodenaufruf beschreibt (die Referenz auf) das Objekt, für das die Methode aufgerufen werden soll, das sogenannte *Empfängerobjekt*. Der Ausdruck kann selbstverständlich zusammengesetzt und geklammert sein (beispielsweise haben wir in Zeile (21) den Ausdruck `System.out` zur Verdeutlichung geklammert). Der Typ des Ausdrucks muss eine Klasse bezeichnen, die eine Methode mit dem angegebenen Methodennamen besitzt. Die aktuelle Parameterliste ist eine Liste von Ausdrücken.

*Empfänger-  
objekt*

Die Auswertung eines Methodenaufrufs terminiert abrupt, wenn die Aus-

wertung des Ausdrucks den Wert `null` ergibt, wenn die Auswertung eines der aktuellen Parameter abrupt terminiert oder wenn die Ausführung des Methodenrumpfes abrupt terminiert.

Entsprechend der Abkürzungsregel beim Attributzugriff kann beim Methodenaufruf die Angabe von `this.` entfallen. Zeile (21) zeigt zwei Methodenaufrufe, wobei `System.out` das Objekt für die Standardausgabe beschreibt.

**Objektorientierte Programme.** In einer Sprache wie Java, die Klassen zur Beschreibung von Objekten verwendet, ist ein objektorientiertes Programm im Wesentlichen eine Ansammlung von Klassen. Ein Programm wird gestartet, indem man eine Methode einer der Klassen aufruft. Ein kleines Problem dabei ist, dass man nach dem bisher Gesagten ein Empfängerobjekt benötigt, um eine Methode aufzurufen, ein solches Objekt beim Programmstart aber noch nicht existiert. Java löst dieses Problem, indem es die Deklaration sogenannter statischer Methoden oder Klassenmethoden unterstützt. Das sind Methoden ohne impliziten Parameter, die man analog zu Prozeduren prozeduraler Sprachen direkt aufrufen kann. Klassenmethoden werden in Unterabschn. 2.1.4.2 behandelt. Für das Verständnis des Folgenden reicht es zu wissen, dass es solche Methoden gibt und dass sie durch Voranstellen des Schlüsselwortes `static` deklariert werden.

In Java gibt es keine als Hauptprogramm ausgezeichneten Programmteile wie etwa in Pascal. Für den Programmstart kann jede Klasse benutzt werden, die eine Klassenmethode mit Namen `main` und einen Parameter vom Typ `String[]` besitzt. Wir betrachten dazu folgendes Beispiel mit den Klassen `Test1` und `Test2`:

```
class Test1 {
    public static void main( String[] args ){
        Test2 obj = new Test2();
        obj.drucke("Ich bin ein Objekt vom Typ Test2");
    }
    public void hallo(){
        System.out.println("Jemand hat mich gegr\u00FC\u00DFt");
    }
}

class Test2 {
    public static void main( String[] args ){
        (new Test1()).hallo();
    }
    public void drucke( String s ) {
        System.out.println( s );
    }
}
```

Befindet sich der Programmtext der Klassen in den Dateien `Test1.java` bzw. `Test2.java`, können die Klassen mit `javac Test1.java` bzw. `javac Test2.java` übersetzt werden. Mit jeder der beiden Klassen kann dann eine Programmausführung gestartet werden. Beispielsweise führt das Kommando `java Test1` zum Aufruf der Methode `main` von Klasse `Test1`. (Als Parameter wird dabei ein Feld ohne Einträge, d.h. mit Länge 0 übergeben. Will man beim Programmstart Parameter mitgeben, muss man sie im Kommando hinter dem Klassennamen angeben; siehe Abschnitt 1.3.3.) Im Rumpf von `Test1.main` wird eine Instanz der Klasse `Test2` erzeugt und der lokalen Variablen `obj` zugewiesen (beachte: Da Klasse `Test2` keinen Konstruktor deklariert, kommt der default-Konstruktor zum Einsatz). Für das neu erzeugte Objekt wird dann die Methode `drucke` der Klasse `Test2` aufgerufen. In entsprechender Weise kann man mit der Klasse `Test2` die Programmausführung beginnen.

Grundsätzlich besteht ein Java-Programm aus einer Ansammlung von Klassen. Wie im Laufe des Kurses noch deutlich werden wird, ist es allerdings nicht immer leicht, präzise zu sagen, welche Klassen zu einem Programm gehören. Dies gilt insbesondere, wenn ein Programm sich erst zur Ausführungszeit entscheidet, welche Klassen es noch dynamisch dazu laden möchte, oder wenn ein Programm auf mehrere Rechner verteilt ist (siehe Kap. 7). Darüber hinaus führt der intensive Gebrauch von Bibliotheksklassen in der objektorientierten Programmierung dazu, den statisch angelegten Programmbegriff aufzuweichen. Vielfach erscheint eine Sichtweise angemessener, in der Klassen als Komponenten von offenen Systemen betrachtet werden, d.h. von Systemen, bei denen die teilnehmenden Komponenten nicht von vornherein festgelegt sind. Jede Klasse sorgt für die Erzeugung ihrer eigenen Objekte und stellt den Programmcode für die zugehörigen Methoden bereit. Sie benötigt andere Klassen, damit sie arbeiten kann. Welche Klassen das sind, lässt sich zur Übersetzungszeit bestimmen. Aber im Allgemeinen steht weder zur Übersetzungszeit, noch beim Systemstart fest, welche Klassen darüber hinaus oder stellvertretend während der Laufzeit des Systems zum Einsatz kommen.

Selbst in dem einfachen, oben angegebenen Beispiel ist nicht ohne weiteres zu erkennen, welche Klassen man zum Programm rechnen soll. Zwar ist klar, dass keine der beiden Klassen für sich genommen ein vollständiges Programm sein kann, da jede Klasse die jeweils andere benutzt: Sie erzeugt ein Objekt der anderen Klasse und ruft darauf eine Methode auf. Aber zu dem Programm gehören auch Klassen der Java-Bibliothek, z.B. die Klassen `String` und `System`, und damit alle transitiv von diesen verwendeten Klassen (beispielsweise die Klasse der Objekte, die von `System.out` referenziert werden; siehe Ausschnitt der Klasse `System` auf Seite S. 94). Da es insbesondere im Zusammenhang mit Subtyping und Vererbung nicht einfach ist, genau zu definieren, wann eine Klasse von einer anderen verwendet wird,

begnügen wir uns im Folgenden mit der groben Festlegung, das ein Java-Programm aus den benutzergeschriebenen und den Bibliotheksklassen besteht.

**Zusammenfassung.** Objekte lassen sich nach den Operationen, die auf ihnen definiert sind, und den Zuständen, die sie einnehmen können, klassifizieren; d.h. alle Objekte mit den gleichen Operationen und der gleichen Zustandsmenge werden in einer Klasse zusammengefasst. Andersherum besehen, legt eine Klasse fest, welche Operationen auf ihren Objekten möglich und zulässig sind und welche Zustände sie einnehmen können. Diese Sichtweise wird von den meisten objektorientierten Programmiersprachen übernommen, d.h. der Programmierer beschreibt nicht die Eigenschaften einzelner Objekte, sondern deklariert Klassen, die die Eigenschaften ihrer Objekte beschreiben, und kann dann Objekte als Instanzen zu den deklarierten Klassen erzeugen.

### 2.1.3 Benutzen und Entwerfen von Klassen

Klassen bilden das zentrale Sprachkonstrukt von Java. Dementsprechend stehen beim Programmmentwurf zwei Fragen im Mittelpunkt:

1. Welche existierenden Klassen können für den Programmmentwurf herangezogen werden?
2. Welche Klassen müssen neu entworfen werden?

Insbesondere die erste Fragestellung gewinnt in der auf Wiederverwendung ausgerichteten objektorientierten Programmierung zunehmend an Bedeutung. Im Laufe dieses Kurses werden wir diesem Aspekt beim Kennenlernen der Java-Bibliothek immer wieder begegnen. Das Zusammenspiel von Benutzung existierender Klassen und Entwurf neuer Klassen soll in diesem Abschnitt an einem kleinen Beispiel betrachtet werden; dabei soll auch gezeigt werden, wie die im letzten Abschnitt vorgestellten Konstrukte im Rahmen einer Entwurfsaufgabe anzuwenden sind.

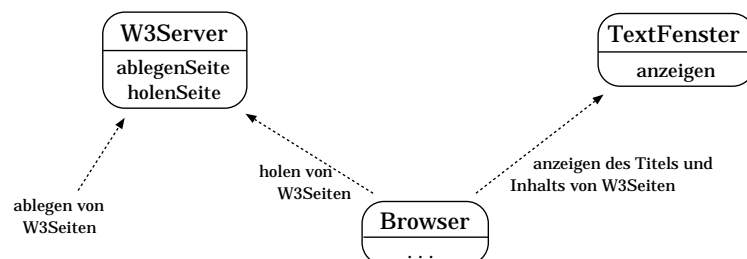


Abbildung 2.2: Zusammenarbeit von Server, Textfenster und Browser

**Aufgabenstellung.** Ein rudimentäres Browser-Programm soll realisiert werden, mit dem Objekte vom Typ `W3Seite` bei einem Server geholt und in einem Fenster angezeigt werden können (vgl. Abb. 2.2). Der Seiten-Server ist ein Objekt, bei dem man W3-Seiten unter einer Adresse ablegen und danach unter dieser Adresse wieder holen kann. Die Adresse ist eine beliebige Zeichenreihe. Hier ist die Klassenschnittstelle des Servers:

```
class W3Server {
    W3Server() { ... }
    void ablegenSeite( W3Seite s, String sadr ) { ... }
    W3Seite holenSeite( String sadr ) { ... }
}
```

Zum Anzeigen von Seiten soll die Klasse `TextFenster` benutzt werden:

```
class TextFenster ... {
    ...
    TextFenster() { ... }
    void anzeigen( String kopfzeile, String text ) { ... }
}
```

Der Konstruktor erzeugt ein `TextFenster`-Objekt, das beim ersten Aufruf der Methode `anzeigen` mit der angegebenen Kopfzeile und dem angegebenen Text auf dem Bildschirm erscheint. Jeder weitere Aufruf von `anzeigen` auf ein `TextFenster`-Objekt führt dazu, dass die Kopfzeile und der Text entsprechend den Parametern geändert werden. Für die Darstellung wird der Text an Wortzwischenräumen geeignet in Zeilen umbrochen. Das Textfenster wird automatisch aktualisiert, wenn es auf dem Bildschirm wieder in den Vordergrund kommt (beispielsweise durch Verschieben eines anderen Fensters, das das Textfenster verdeckt hat).

**Entwerfen eines Browsers.** Browser mit dem beschriebenen Verhalten sollen durch eine Klasse `Browser` implementiert werden (vgl. Abb. 2.3).

Bei Erzeugung eines `Browser`-Objekts wird eine Referenz auf den zuständigen Server übergeben. Jedes `Browser`-Objekt besitzt ein `Textfenster` zum Anzeigen der aktuellen Seite. Nachdem das `Textfenster` erzeugt ist, lädt der `Browser` eine Startseite und startet die interaktive Steuerung des Browsers.

Die Steuerung erlaubt es dem Benutzer, interaktiv neue Seiten zu laden, indem er deren Adresse eingibt, oder den `Browser` zu beenden. Eine Implementierung der interaktiven Steuerung behandeln wir in Abschn. 2.1.4 zur Illustration weiterer sprachlicher Konstrukte von Java.

Abbildung 2.5 zeigt einen Testrahmen für die beschriebenen Klassen. Die vier Beispielseiten entsprechen den Anfängen der Kapitel 2 und 3 und der Abschnitte 2.1 und 2.2. Der Operator „+“ bezeichnet dabei die *Konkatenation von Zeichenreihen*. Die Zeichenreihe "<BR>" wird zur Markierung von

*Konkatenation  
von Zeichen-  
reihen*



```

class Browser {
    W3Server    meinServer;
    TextFenster oberfl;
    W3Seite     aktSeite; // aktuelle Seite

    Browser( W3Server server ){
        meinServer = server;
        oberfl      = new TextFenster();
        laden( new W3Seite("Startseite",
                           "NetzSurfer: Keiner ist kleiner") );
        interaktiveSteuerung();
    }
    void laden( W3Seite s ){
        aktSeite = s;
        oberfl.anzeigen( aktSeite.getTitel(),
                         aktSeite.getInhalt());
    }
    void interaktiveSteuerung() { ... }
}

```

Abbildung 2.3: Implementierung eines rudimentären Browsers

HTML

Zeilenumbrüchen verwendet (die Syntax mit den spitzen Klammern ist der Seitenbeschreibungs-Sprache *HTML* – **H**ypertext **M**arkup **L**anguage – entlehnt; BR steht für **B**reak **R**ow). Abbildung 2.4 zeigt die Seite *kap2* dargestellt in einem Textfenster. In der bisher behandelten Ausbaustufe des Browsers muss der Benutzer die Referenzen auf andere Seiten innerhalb des Texts (im Beispiel sind das *kap3*, *abs2.1* und *abs2.2*) noch von Hand in den Browser eingeben, um diese Seiten zu laden.

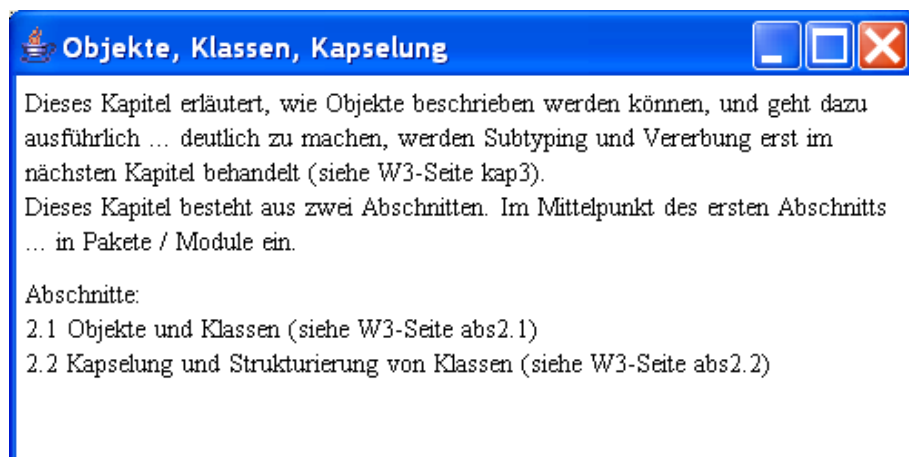


Abbildung 2.4: Textfenster mit Seite „kap2“

```

class Main {
    public static void main( String[] args ){
        W3Seite kap2  = new W3Seite(
            "Objekte, Klassen, Kapselung",
            "Dieses Kapitel erl\u00E4utert, wie Objekte "
            +" beschrieben werden k\u00F6nnen, und geht dazu "
            +" ausf\u00Fchrlich ... deutlich zu machen, "
            +" werden Subtyping und Vererbung erst im n\u00E4chsten "
            +" Kapitel behandelt (siehe W3-Seite kap3). <BR> "
            +"Dieses Kapitel besteht aus zwei Abschnitten. Im "
            +" Mittelpunkt des ersten Abschnitts ... in Pakete /"
            +" Module ein. <BR> <BR> "
            +"Abschnitte: <BR> "
            +"2.1 Objekte und Klassen (siehe W3-Seite abs2.1) <BR> "
            +"2.2 Kapselung und Strukturierung von Klassen (siehe "
            +" W3-Seite abs2.2) "
        );
        W3Seite kap3  = new W3Seite(
            "Vererbung und Subtyping",
            "Dieses Kapitel ... "
        );
        W3Seite abs21 = new W3Seite(
            "Objekte und Klassen",
            "Dieser Abschnitt behandelt die Frage: Was ... "
        );
        W3Seite abs22 = new W3Seite(
            "Kapselung und Strukturierung von Klassen",
            "Der vorangegangene Abschnitt ... "
        );

        W3Server testServer = new W3Server();
        testServer.ablegenSeite(kap2, "kap2");
        testServer.ablegenSeite(kap3, "kap3");
        testServer.ablegenSeite(abs21, "abs2.1");
        testServer.ablegenSeite(abs22, "abs2.2");

        new Browser( testServer );
    }
}

```

Abbildung 2.5: Klasse mit Programmeinstiegspunkt für die Browser-Klasse

### 2.1.4 Weiterführende Sprachkonstrukte

Dieser Abschnitt behandelt Erweiterungen des in 2.1.2 vorgestellten Java-Sprachkerns. Im Einzelnen stellen wir das Initialisieren von Attributen und Variablen, das Überladen von Methodennamen sowie Klassenmethoden und Klassenattribute vor. Begleitend dazu werden wir erste Einblicke in die vordefinierten Java-Klassen `String` und `System` geben und das Browserbeispiel weiter vorantreiben.

#### 2.1.4.1 Initialisierung und Überladen

Dieser Unterabschnitt zeigt zunächst, wie Variablen und Attribute bei der Deklaration initialisiert werden können und stellt dann Initialisierungsblöcke vor. Anschließend wird das Überladen von Methodennamen erläutert.

**Initialisieren von Variablen und Attributen.** Lokale Variablen und Attribute können an ihrer Deklarationsstelle initialisiert werden. Beispielsweise ist das Programmfragment

```
class C {
    int ax, ay;
    C(){
        ax = 7;  ay = 9;    m();
    }
    void m(){
        int v1, v2;
        v1 = 4848;  v2 = -3;  ...
    }
}
```

äquivalent zu der folgenden Klassendeklaration:

```
class C {
    int ax = 7, ay = 9;
    C(){
        m();
    }
    void m(){
        int v1 = 4848, v2 = -3;  ...
    }
}
```

Insbesondere ist aus dem Beispiel zu erkennen, dass die Initialisierung von Attributen vor der Ausführung der entsprechenden Konstruktorrumpfe vorgenommen wird.

Java bietet die Möglichkeit Variablen und Attribute durch Voransetzen des Schlüsselworts `final` als *unveränderlich* zu deklarieren. Oft bezeichnet man solche Variablen bzw. Attribute auch als *benannte Konstanten*. Unveränderliche Variablen und Attribute müssen entweder an der Deklarationsstelle oder im umfassenden Block bzw. dem Konstruktor initialisiert werden:

```
class C {
    final int ax, ay = 9;
    C() {
        ax = 7;    // KORREKT: Initialisierung im Konstruktor
                  // und nicht an Deklarationsstelle

        m();
    }
    void m() {
        final int v1 = 4848, v2 = -3;
        v2 = 0;    // UNZULAESSIG: v2 ist unveraenderlich
        ay = 34;  // UNZULAESSIG: ay ist unveraenderlich
        ...
    }
}
```

Wenn Variablen und Attribute während der Programmausführung nicht verändert werden müssen, sollten sie auch als unveränderlich deklariert werden. Das vermeidet Programmierfehler und gestattet dem Übersetzer im Allgemeinen, effizienteren Code zu erzeugen.

**Initialisierungsblöcke.** In gleicher Weise wie im vorigen Abschnitt beschrieben können auch Klassenattribute (siehe Abschnitt 2.1.4.2) direkt an der Deklarationsstelle initialisiert werden. Häufig aber sind einfache Initialisierungsausdrücke wie die im vorigen Abschnitt gezeigten nicht hinreichend, um die gewünschte Initialisierung zu erreichen. Die Möglichkeit zur Initialisierung von Feldvariablen mit Hilfe von Feldinitialisierern, wie wir sie in Abschnitt 1.3.1.2 in Abbildung 1.7 bereits kennengelernt haben (z.B. `Fliegen = {'F','l','i','e','g','e','n'};`), ist oft nicht ausreichend.

Für komplexe Initialisierungsaufgaben stellt Java sog. *Initialisierungsblöcke* bereit: *Statische* Initialisierungsblöcke für Klassenattribute und *nicht-statische* Initialisierungsblöcke für Instanzattribute. Syntaktisch handelt es sich bei statischen und nicht-statischen Initialisierungsblöcken um in geschweifte Klammern eingeschlossene Anweisungsblöcke, die im Falle statischer Initialisierungsblöcke durch das Schlüsselwort `static` eingeleitet werden, das im Falle nicht-statischer Initialisierungsblöcke entfällt.

Statische und nicht-statische Initialisierungsblöcke können in einer Klasse an allen Stellen stehen, an denen auch eine Attribut- oder Methodendeklaration stehen könnte. Insbesondere kann es mehrere statische und nicht-statische Initialisierungsblöcke in einer Klasse geben. Initialisierungsblöcke

*Initialisierungs-  
block*

*benannte  
Konstante*

selbst können grundsätzlich beliebige Anweisungen enthalten, auch Aufrufe von Methoden in anderen Klassen. Allerdings ist wie im Fall der im vorigen Abschnitt beschriebenen einfachen Initialisierungen darauf zu achten, dass nicht auf Variablen zugegriffen wird, die textuell erst später deklariert werden. Auch darf der Aufruf von Methoden in anderen Klassen und die damit ggf. verbundene Ausführung ihrer Initialisierungsblöcke zu keinen zyklischen Abhängigkeiten führen.

Statische wie nicht-statische Initialisierungsblöcke werden dabei zusammen mit den im vorigen Abschnitt beschriebenen Initialisierungen in der Reihenfolge ausgeführt, in der sie in der Klasse aufgeschrieben sind. Statische Initialisierungsblöcke werden dabei genau einmal ausgeführt, nämlich beim ersten Laden ihrer Klasse, nicht-statische bei jedem Instanziiieren ihrer Klasse, und zwar vor der Ausführung des aufgerufenen Konstruktors. Das folgende Beispiel zeigt eine Anwendung von Initialisierungsblöcken. Dabei nehmen wir an, dass die Methode `fibonacci` in einer hier nicht aufgeführten, aber von `D` aus sichtbaren Klasse `Fib` als Klassenmethode deklariert (siehe Abschnitt 2.1.4.2) ist <sup>6</sup>.

```
class D {
    static int[] f = new int[10];

    // Statischer Initialisierungsblock
    static {
        System.out.println("Initialisierung von f beginnt.");
        for (int i=1; i<f.length; i++)
            f[i] = Fib.fibonacci(i);
        System.out.println("Initialisierung von f abgeschlossen.");
    }

    int eins = 1;
    int[] g = new int[5];

    // Nicht-statischer Initialisierungsblock
    {
        g[0] = eins;
        for (int j=1; j<g.length; j++) {
            int h = 1;
            for (int k=1; k <= j; k++)
                h = h*k;
            g[j] = h;
        }
    }
}
```

---

<sup>6</sup>Der Ausdruck `i++`, der oft in Zählschleifen verwendet wird, erhöht ebenso wie der Ausdruck `++i` als Seiteneffekt den Wert der Variablen `i` um 1. Der Unterschied dieser beiden Ausdrücke ist, dass die Auswertung von `i++` als Ergebnis den Wert von `i` *vor* und `++i` den Wert von `i` *nach* der Erhöhung um 1 zurückliefert. Im Gegensatz zu `i++` ist `++i` also äquivalent zu `i=i+1`.

```

    }
}
}

```

Wie oben angedeutet und im Beispiel gezeigt, können mit Initialisierungsblöcken komplexe Initialisierungsaufgaben bewältigt werden. Im Fall von nicht-statischen Initialisierungsblöcken können sie auch zur besseren Lesbarkeit des Programms beitragen, da sie erlauben, den Initialisierungscode in unmittelbarer Nähe der Deklarationsstelle zu platzieren und nicht abgesetzt in den einzelnen Konstruktordeklarationen. Nicht-statische Initialisierungsblöcke bieten sich darüberhinaus auch zur Ausführung allen Konstruktoren einer Klasse gemeinsamen Initialisierungscodes an und damit als Alternative zu einem von allen „geteilten“ Konstruktor.

**Überladen von Methodennamen und Konstruktoren.** In Java ist es erlaubt, innerhalb einer Klasse mehrere Methoden mit dem gleichen Namen zu deklarieren. Eine derartige Mehrfachverwendung nennt man *Überladen* eines Namens. Methoden mit gleichen Namen müssen sich in der Anzahl oder in den Typen der Parameter unterscheiden. Dadurch ist es dem Übersetzer möglich, die Überladung *aufzulösen*, d.h. für jede Aufrufstelle zu ermitteln, welche von den Methoden gleichen Namens an der Aufrufstelle gemeint ist. Der Rückgabotyp einer Methode wird zur Auflösung nicht herangezogen mit einer Ausnahme: einer der Parametertypen ist durch Einsetzen eines Typparameters aus einem parametrischen Typ, auch *generischer Typ* genannt, hervorgegangen. Diesen Fall behandeln wir in Abschnitt 2.1.6.2 im Kontext parametrischer Klassen. Entsprechend dem Überladen von Methodennamen unterstützt Java auch das Überladen von Konstruktoren.

Überladen

auflösen

Gute Beispiele für das Überladen findet man in der Klasse `String` des Standard-Pakets<sup>7</sup> `java.lang` der Java-Bibliothek. Ein `String`-Objekt besitzt in Java drei Attribute: eine Referenz auf ein `char`-Feld, den Index des Feldelements mit dem ersten gültigen Zeichen und die Anzahl der Zeichen der repräsentierten Zeichenreihe. Die Klasse `String` hat elf Konstruktoren und 47 Methoden. Wir betrachten hier nur einen kleinen Ausschnitt mit vier Konstruktoren und sieben Methoden<sup>8</sup>:

```

class String {
    /** The value is used for character storage */
    char[] value;
    /** The offset is the first index of the used storage */
    int offset;
    /** The count is the number of characters in the String */

```

<sup>7</sup>Der Begriff des Pakets wird in Abschnitt 2.2.2 eingeführt.

<sup>8</sup>Gegenüber dem originalen Quellcode haben wir auch die Zugriffsmodifikatoren weggelassen (vgl. Abschn. 2.2).

```

int count;

String() { value = new char[0]; }
String( String value ) { ... }
String( char[] value ) {
    this.count = value.length;
    this.value = new char[count];
    System.arraycopy(value, 0, this.value, 0, count);
}
String( char[] value, int offset, int count ) { ... }

int indexOf(int ch) { return indexOf(ch, 0); }
int indexOf(int ch, int fromIndex) { ... }
int indexOf(String str) { return indexOf(str, 0); }
int indexOf(String str, int fromIndex) { ... }

int length() {
    return count;
}
char charAt(int index) {
    if ((index < 0) || (index >= count)) {
        throw new StringIndexOutOfBoundsException(index);
    }
    return value[index + offset];
}
boolean equals(Object anObject) { ... }
}

```

Der Konstruktor mit leerer Parameterliste liefert ein `String`-Objekt, das die leere Zeichenreihe repräsentiert. Der Konstruktor mit dem Parameter vom Typ `String` liefert ein neues `String`-Objekt, das die gleiche Zeichenreihe wie der Parameter repräsentiert. Der dritte Konstruktor erzeugt ein neues `String`-Objekt mit den im übergebenen Feld enthaltenen Zeichen (auf die Methode `arraycopy` gehen wir im folgenden Unterabschnitt kurz ein). Mit dem vierten Konstruktor kann ein neues `String`-Objekt zu einem Ausschnitt aus einem übergebenen Feld erzeugt werden; der Ausschnitt wird durch den `offset` des ersten Zeichens und die Anzahl der Zeichen (`count`) festgelegt.

Die Methoden mit Namen `indexOf` bieten vier Möglichkeiten, den Index des ersten Vorkommens eines Zeichens zu ermitteln. Die erste Version beginnt mit der Suche am Anfang der Zeichenreihe, die zweite beim angegebenen Index und die dritte mit Beginn des ersten Vorkommens der angegebenen Teilzeichenreihe `str`; die vierte Version verallgemeinert die dritte, indem sie mit der Suche der Teilzeichenreihe `str` erst ab Index `fromIndex` beginnt. Das Beispiel zeigt sehr schön die Anwendung von Überladen. Alle Methoden erfüllen eine ähnliche Aufgabe, sodass es gerechtfertigt ist, ihnen

den gleichen Namen zu geben. Sie unterscheiden sich nur darin, wie allgemein sie sind (die zweite bzw. vierte Version sind allgemeiner als die erste bzw. dritte Version) und welche Parameter sie benötigen.

Die Methoden `length`, `charAt` und `equals` werden wir im Folgenden benötigen. `length` gibt die Anzahl der im `String`-Objekt gespeicherten Zeichen zurück. `charAt` liefert das Zeichen, das an der durch `index` bezeichneten Position in der Zeichenkette zu finden ist. Die Methode `equals` dient im Wesentlichen dazu, zwei `String`-Objekte daraufhin zu vergleichen, ob sie die gleiche Zeichenreihe repräsentieren. (Beachte: Zeichenreihengleichheit kann nicht mit dem Operator `==` geprüft werden, da dieser vergleicht, ob die `String`-Objekte *identisch* sind; d.h. es wird geprüft, ob die Ausdrücke auf beiden Seiten des `==`-Operators nach ihrer Auswertung dasselbe `String`-Objekt referenzieren. Es kann aber z.B. zwei verschiedene `String`-Objekte geben, die die gleiche Zeichenreihe repräsentieren.)

#### 2.1.4.2 Klassenmethoden und Klassenattribute

Attribute und Methoden können in Java nicht nur Objekten zugeordnet werden, sondern auch Klassen. Was das im Einzelnen bedeutet, wie es deklariert wird und für welche Anwendungen dieses Konstrukt sinnvoll ist, soll im Folgenden kurz erläutert werden.

**Deklaration und Bedeutung.** *Klassenmethoden* bzw. *Klassenattribute* werden deklariert, indem man eine Methoden- bzw. Attributdeklaration mit dem Schlüsselwort `static` beginnt; dementsprechend werden sie häufig auch als *statische* Methoden bzw. Attribute bezeichnet. Diese Bezeichnung ist insofern berechtigt, als dass Klassenmethoden statisch, also zur Übersetzungszeit gebunden werden und nicht dynamisch, wie es die Regel bei normalen Methoden ist (zur dynamischen Bindung vgl. z.B. S. 46). Für Klassenattribute gilt, dass sie nicht dynamisch bei der Objekterzeugung alloziert werden; vielmehr wird statisch vom Übersetzer Speicherplatz für sie vorgesehen.

*Klassenmethode*  
*Klassenattribut*

Wie wir gesehen haben, besitzt jedes Objekt für jedes normale Attribut seiner Klasse eine objektlokale Variable (vgl. die Bemerkungen zum Begriff „Instanzvariable“ auf S. 78). Im Gegensatz dazu entspricht jedem Klassenattribut genau eine Variable, unabhängig davon, wieviele Objekte der Klasse erzeugt werden; insbesondere existiert diese Variable auch, wenn gar kein Objekt der Klasse erzeugt wird. Für den Zugriff auf Klassenattribute wird die Syntax `Klassenname . Attributname` verwendet; innerhalb der Klassendefinition reicht die Angabe des Attributnamens.

Klassenmethoden ähneln Prozeduren der prozeduralen Programmierung. Sie besitzen keinen impliziten Parameter. Deshalb macht es auch keinen Sinn, innerhalb von Klassenmethoden auf die normalen Attribute einer Klasse zuzugreifen; denn erstens wäre nicht klar, zu welchem Objekt die Attribute



gehören sollen, auf die der Zugriff erfolgt, und zweitens sollen Klassenmethoden auch ausführbar sein, wenn kein Objekt der Klasse existiert. Selbstverständlich darf eine Klassenmethode auf die zugehörigen Klassenattribute zugreifen. Ein typisches Beispiel für eine Klassenmethode ist die Methode mit Namen `main`, die zu jedem ausführbaren Programm gehört (vgl. S. 82). Besonderheiten bei der Deklaration von Klassenmethoden, die zu parametrischen Klassen gehören, werden wir in Abschnitt 2.1.6.2 behandeln.

**Anwendungsbeispiele aus dem Standard-Paket.** Klassenattribute und -methoden werden häufig verwendet, um Informationen über alle Objekte einer Klasse zentral bei der Klasse zu verwalten oder um prozedurale Aspekte in objektorientierten Programmen zu realisieren. Beispielsweise besitzt die Klasse `String` neun Klassenmethoden mit Namen `valueOf`, die u.a. zu den Werten der Basisdatentypen eine geeignete Zeichenreihenrepräsentation liefern; hier sind die Deklarationen von zweien dieser Methoden:

```
static String valueOf( long l ) { ... }
static String valueOf( float f ) { ... }
```

Der Aufruf `String.valueOf( (float)(7./9.) )` liefert zum Beispiel die Zeichenreihe `"0.7777778"` und demonstriert, wie Klassenmethoden aufgerufen werden. Wiederum gilt, dass innerhalb der Klassendefinition die Angabe des Klassennamens beim Aufruf statischer Methoden entfallen kann.

Eine andere Anwendung von Klassenmethoden und -attributen ist die Realisierung von Modulen, wie sie in prozeduralen Sprachen üblich sind. Klassenattribute entsprechen dabei modullokalen Variablen, Klassenmethoden übernehmen die Rolle von modullokalen Prozeduren. Ein typisches Beispiel dafür ist die Klasse `System` der Java-Bibliothek. Sie beschreibt einen wichtigen Teil der Systemschnittstelle von Java-Programmen. Hier ist ein kurzer Ausschnitt ihrer Klassendeklaration:

```
class System {
    final static InputStream in = ...;
    final static PrintStream out = ...;
    static void exit(int status) { ... }
    static native void arraycopy(Object src,int src_position,
                                Object dst,int dst_position, int length);
}
```

Die Klassenattribute `in` und `out` ermöglichen den Zugriff auf den Eingabe- bzw. Ausgabestrom eines Programmlaufs. Der Zugriff auf den Ausgabestrom wurde ja bereits mehrfach verwendet, indem wir für das Ausgabestrom-Objekt die Methode `print` aufgerufen haben:

```
System.out.print("Das klaert die Syntax des Printaufrufs");
```

Die Methode `exit` ermöglicht es, die Ausführung eines Java-Programms direkt abzubrechen (üblicher aktueller Parameter für `status` ist 0). Die Methode `arraycopy` ist eine System-Prozedur zum effizienten Kopieren von Feldern (sie ist uns schon bei der Implementierung eines der String-Konstrukturen begegnet, siehe S. 92). Sie kopiert die Elemente des Feldes `src` mit den Indizes `src_position` bis `src_position+length-1` in das Feld `dst` und zwar ab Index `dst_position`. Das Schlüsselwort `native` bei der Deklaration von `arraycopy` besagt, dass die Implementierung nicht in Java selbst erfolgt, sondern in einer anderen Programmiersprache (zur Zeit werden Programmierschnittstellen zu C und C++ unterstützt).

### 2.1.4.3 Zusammenwirken der Spracherweiterungen

Das Browserbeispiel soll uns hier dazu dienen, das Zusammenwirken der behandelten Spracherweiterungen in einem größeren Programmkontext zu studieren. Insbesondere werden wir sehen, wie statische Attribute und Methoden genutzt werden können, um Informationen, die allen Objekten einer Klasse gemeinsam sind, innerhalb der Klasse zu verwalten. Das spart nicht nur Speicherplatz, sondern ermöglicht auch eine Koordinierung des Verhaltens der Objekte.

Im Prinzip gestattet es die Browser-Klasse von S. 86, innerhalb einer Programmausführung mehrere Browser-Objekte zu starten. Insbesondere in realistischen Umgebungen ist dies sinnvoll, da man dann in einem Browserfenster eine Seite betrachten kann, während das andere Browser-Objekt eine neue Seite lädt. Allerdings bringt die interaktive Steuerung über die Standard-Eingabe und -Ausgabe die Einschränkung mit sich, dass immer nur einer der aktiven Browser gesteuert werden kann. (In Kap. 5 werden wir zeigen, wie diese Einschränkung mittels interaktiver graphischer Bedienoberflächen überwunden werden kann.)

Die Browser-Klasse von Abschn. 2.1.3 ermöglicht zwar das Erzeugen mehrerer Browser-Objekte, unterstützt diese aber nicht durch eine gemeinsame interaktive Steuerung. Jedes Browser-Objekt besitzt seine eigene interaktive Steuerung (siehe Abbildung 2.3 auf Seite 86). Dies führt dazu, dass jeweils nur der zuletzt gestartete Browser bedient werden kann – eine untragbare Einschränkung. Diese Einschränkung überwinden wir durch folgendes Redesign der Klasse `Browser`:

1. Wie in der Browser-Klasse von S. 86 bekommt jedes Browser-Objekt eine eigene Oberfläche und eine aktuelle Seite. Der zugehörige W3-Server wird allerdings in einem statischen Attribut gespeichert, da wir annehmen, dass er in allen Browsern einer Programmausführung gleich ist.
2. Um mit der interaktiven Steuerung zwischen den gestarteten Browsern wechseln zu können, verwalten wir die gestarteten Browser mit statischen Attributen in der Klasse `Browser`: Außer einem Feldattribut für

die gestarteten Browser gibt es Klassenattribute für die maximal mögliche Anzahl gestarteter Browser, für den nächsten freien Index im Feld und für den Index des aktuellen Browsers, d.h. des Browsers, der aktuell gesteuert werden soll.

3. Die Referenz auf die Startseite speichern wir in einem statischen Attribut, damit sie für alle Browser genutzt werden kann.
4. Wir sehen zwei Konstruktoren vor: Der eine Konstruktor dient zum Erzeugen des ersten Browser-Objekts, der andere zum Erzeugen weiterer Browser. Den gemeinsamen Teil der Konstruktoren implementieren wir als Hilfsmethode namens `initialisieren`.
5. Die interaktive Steuerung realisieren wir als Klassenmethode.
6. Um die Anwendung zu vereinfachen, sehen wir eine statische Methode `start` vor, die den richtigen Konstruktor aufruft und dann die interaktive Steuerung startet<sup>9</sup>.

Abbildung 2.6 zeigt die so überarbeitete Browserklasse, die nochmals die in diesem Abschnitt eingeführten Spracherweiterungen illustriert: Initialisieren von Attributen, unveränderliche Attribute, Felder, Überladen von Konstruktoren sowie Klassenattribute und -methoden kombiniert mit normalen Attributen und Methoden. Die Implementierung der interaktiven Steuerung ist in Abb. 2.7 aufgeführt. Wir wollen sie uns im Folgenden etwas genauer anschauen, auch um bei der Behandlung graphischer Bedienoberflächen in Kap. 5 auf ein Beispiel für eine Menü-geführte Steuerung zurückgreifen zu können.

Die interaktive Steuerung liest von der Eingabekonzole. Sie versteht vier Steuerzeichen: *l* zum Laden einer neuen Seite im aktuellen Browser – der Name der Seite wird dann vom Benutzer abgefragt; *n* zum Starten eines neuen, weiteren Browsers; *w* zum Wechseln des aktuellen Browsers – gewechselt wird zum Browser mit dem nächsten Index bzw. zum Browser mit dem ersten Index; *e* zum Beenden des Browserprogramms. Für die Ein- und Ausgabe von Zeichenreihen nutzt die interaktive Steuerung die Klasse `Konsole` mit den Klassenmethoden `readString` und `writeString`:

```
class Konsole {
    static String readString() { ... }
    static void writeString( String s ) { ... }
}
```

Als Klasse mit Programmeinstiegspunkt kann wieder die Klasse `Main` aus Abb. 2.5, S. 87, dienen. Dabei muss allerdings der Aufruf des Konstruktors in

<sup>9</sup>Auf diese Weise vermeiden wir auch den schlechten Stil, in einem Konstruktor eine interaktiv arbeitende, nichtterminierende Methode aufzurufen.

```

class Browser {
    TextFenster oberfl;
    W3Seite      aktSeite;
    static W3Server  meinServer;
    static final int MAX_ANZAHL = 4;
    static Browser[] gestarteteBrowser= new Browser[MAX_ANZAHL];
    static int      naechsterFreierIndex = 0;
    static int      aktBrowserIndex;
    static W3Seite  startseite =
        new W3Seite("Startseite","NetzSurfer: Keiner ist kleiner");

    // Konstruktor zum Starten des ersten Browsers
    Browser( W3Server server ) {
        if( naechsterFreierIndex != 0 ) {
            System.out.println("Es sind bereits Browser gestartet");
        } else {
            meinServer = server;
            initialisieren();
        }
    }
    // Konstruktor zum Starten weiterer Browser
    Browser() {
        if( naechsterFreierIndex == MAX_ANZAHL ) {
            System.out.println("Maximale Anzahl Browser erreicht");
        } else {
            initialisieren();
        }
    }
    static void start( W3Server server ) {
        new Browser(server);
        Browser.interaktiveSteuerung();
    }
    void initialisieren() {
        oberfl      = new TextFenster();
        gestarteteBrowser[ naechsterFreierIndex ] = this;
        aktBrowserIndex = naechsterFreierIndex;
        naechsterFreierIndex++;
        laden( startseite );
    }
    void laden( W3Seite s ){
        aktSeite = s;
        oberfl.anzeigen(aktSeite.getTitel(),aktSeite.getInhalt());
    }
    static void interaktiveSteuerung() {
        ... /* siehe folgende Abbildung */ }
}

```

Abbildung 2.6: Browser-Klasse nach Redesign

```
static void interaktiveSteuerung() {
    char steuerzeichen = '0';
    do {
        Konsole.writeString("Steuerzeichen eingeben [lnwe]: ");
        try {
            String eingabe = Konsole.readString();
            if( eingabe.equals("") )
                steuerzeichen = '0';
            else
                steuerzeichen = eingabe.charAt(0);
        } catch( Exception e ) {
            System.exit( 0 );
        }
        switch( steuerzeichen ){
            case 'l':
                String seitenadr;
                Konsole.writeString("Seitenadresse eingeben: ");
                seitenadr = Konsole.readString();
                gestarteteBrowser[aktBrowserIndex] .
                    laden( meinServer.holenSeite( seitenadr ) );
                break;
            case 'n':
                new Browser();
                break;
            case 'w':
                aktBrowserIndex =
                    (aktBrowserIndex+1) % naechsterFreierIndex;
                break;
            case 'e':
                System.exit( 0 );
            default:
                Konsole.writeString("undefiniertes Steuerzeichen\n");
        }
    } while( true );
}
```

Abbildung 2.7: Browsersteuerung über die Konsole

der letzten Zeile der `main`-Methode durch den Aufruf der statischen Methode `start` ersetzt werden, also durch die Anweisung

```
Browser.start(testServer);
```

### 2.1.5 Rekursive Klassendeklaration

Eine Definition nennt man *rekursiv*, wenn die definierte Größe im definierenden Teil vorkommt. Beispielsweise ist eine Methode *m* rekursiv definiert, wenn *m* in ihrem eigenen Rumpf aufgerufen wird; man sagt dann auch einfach, dass *m* rekursiv ist. Entsprechend dem zugrunde liegenden Programmierparadigma (vgl. Abschn. 1.2) unterstützen moderne Programmiersprachen rekursive Definitionen bzw. Deklarationen von Prozeduren/Funktionen/Prädikaten/Methoden und Datentypen. Rekursive Datentypdeklarationen sind bei vielen prozeduralen und objektorientierten Programmiersprachen allerdings nur unter Verwendung expliziter Zeigertypen möglich. Dieser Abschnitt erläutert an einem Beispiel, wie problemlos der Umgang mit rekursiven Klassen in Java ist.<sup>10</sup> Als Beispiel betrachten wir eine Realisierung doppeltverketteter Listen. Damit verfolgen wir drei Ziele: Erstens soll eine kanonische und wichtige rekursive Datentypimplementierung vorgestellt werden; zweitens soll in das Java-Paket `java.util` eingeführt werden; und drittens schaffen wir uns damit Anschauungsmaterial für Abschn. 2.2.

*rekursiv*

Bei der Klasse `Browser` von Abb. 2.6 haben wir bereits von rekursiven Definitionen Gebrauch gemacht, ohne es zu erwähnen: Das Klassenattribut `gestarteteBrowser` ist vom Typ `Browser[]`, d.h. der von der Klasse deklarierte Typ wird innerhalb des Rumpfes der Klasse verwendet. In dem Beispiel können allerdings keine rekursiven Abhängigkeiten zwischen `Browser`-Objekten entstehen: Es gibt keine Kette von Referenzen von einem `Browser`-Objekt zu einem anderen. Wie wir im Laufe des Kurses sehen werden, sind rekursive Abhängigkeiten in der objektorientierten Programmierung eher der Normalfall als die Ausnahme.

**Doppeltverkettete Listen.** Ein *Behälter* (engl. *container*) ist ein Datentyp zum Speichern von Elementen bzw. Objekt-Referenzen. Eine Liste ist ein Behälter, in dem ein Element mehrfach enthalten sein kann und in dem die Elemente geordnet sind; d.h. u.a., dass es sinnvoll ist, nach dem ersten und letzten Element zu fragen und die Elemente der Reihe nach zu durchlaufen. Listen implementiert man üblicherweise mit Hilfe von Feldern, durch einfache Verkettung der Elemente oder durch doppelte Verkettung.

*Behälter*

Wir betrachten hier zunächst einen vereinfachten Ausschnitt aus der Klasse `LinkedList` des Bibliothekspakets `java.util` der Version 1.4<sup>11</sup>, die

<sup>10</sup>Dies liegt daran, dass Objektvariablen in Java automatisch Zeigervariablen sind; Objektvariablen enthalten Referenzen auf Objekte, nicht die Objekte selbst.

<sup>11</sup>Die Version 5.0 verwendet bereits eine parametrische Variante dieser Klasse, wie wir sie

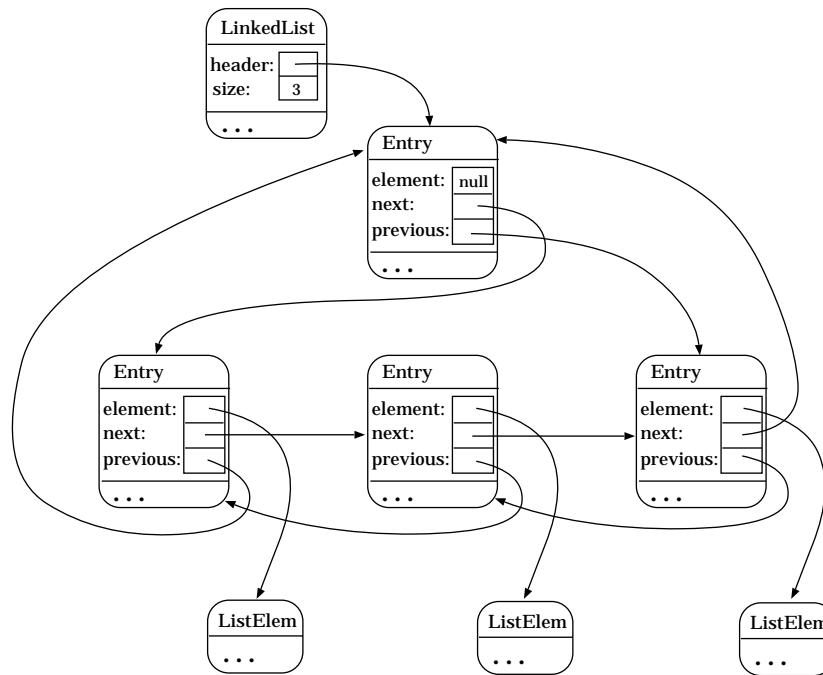


Abbildung 2.8: Doppeltverkettete Liste

doppeltverkettete Listen realisiert. Doppeltverkettete Listen ermöglichen effizientes Arbeiten am Anfang und Ende der Liste (z.B. das Anfügen von Elementen) und das flexible Durchwandern der Liste. Das Objektgeflecht zur Repräsentation einer drei-elementigen doppelt verketteten Liste ist in Abb. 2.8 dargestellt. Die entsprechenden Klassendeklarationen bietet Abbildung 2.9. Dabei gehen wir zunächst davon aus, dass Listenelemente vom Typ `ListElem` sind. Wie dieser Typ implementiert ist, spielt hier keine Rolle. (Eine mögliche Implementierung ist auf S. 117 dargestellt.)

Die Klasse `Entry` zur Beschreibung der Verkettungselemente bietet ein typisches Beispiel für eine direkt rekursive Klassendeklaration: `Entry`-Objekte besitzen Attribute, die direkt `Entry`-Objekte referenzieren.

### 2.1.6 Typkonzept und Parametrisierung von Klassen

In typisierten Programmiersprachen wird den deklarierten Programmelementen ein Typ zugeordnet, der charakterisiert, welche Operationen mit dem Sprachelement zulässig sind (vgl. dazu Abschn. 1.3.1, insbesondere die S. 28 ff). Der Typ eines Objekts gibt darüber Auskunft, welche Attribute das Objekt besitzt und welche Signatur seine Methoden haben. Jede Klasse in Java deklariert einen neuen Typ. Aus der Klassendeklaration kann man die Attribute und deren Typen sowie die erweiterten Methodensignaturen ablesen.

---

später noch kennen lernen werden.

```
class Entry {
    ListElem element;
    Entry next;
    Entry previous;

    Entry(ListElem element, Entry next, Entry previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
    }
}

class LinkedList {
    Entry header = new Entry(null, null, null);
    int size = 0;

    /* Constructs an empty Linked List. */
    LinkedList() {
        header.next = header;
        header.previous = header;
    }
    /* Returns the last Element in this List. */
    ListElem getLast() {
        if( size==0 ) throw new NoSuchElementException();
        return header.previous.element;
    }
    /* Removes and returns the last Element from this List. */
    ListElem removeLast() {
        Entry lastentry = header.previous;
        if(lastentry==header) throw new NoSuchElementException();
        lastentry.previous.next = lastentry.next;
        lastentry.next.previous = lastentry.previous;
        size--;
        return lastentry.element;
    }
    /* Appends the given element to the end of this List. */
    void addLast(ListElem e) {
        Entry newEntry = new Entry(e, header, header.previous);
        header.previous.next = newEntry;
        header.previous = newEntry;
        size++;
    }
    /* Returns the number of elements in this List. */
    int size() {
        return size;
    }
}
```

Abbildung 2.9: Die Klassen Entry und LinkedList



In Kap. 3 werden wir sehen, dass zu dem Typ, den eine Klasse *K* deklariert, nicht nur die Objekte der Klasse *K* gehören, sondern z.B. auch alle Objekte von Klassen, die von *K* erben.

Die Typisierung ermöglicht es, zur Übersetzungszeit die Korrektheit von Attributzugriffen und Methodenaufrufen zu überprüfen. Betrachten wir dazu die letzte Zeile der Methode `removeLast` von Abb. 2.9:

```
return lastentry.element;
```

Gemäß Typisierung kann die Variable `lastentry` nur Referenzen auf Objekte vom Typ `Entry` speichern oder aber die spezielle Referenz `null`, die zu allen Typen gehört. Zur Ausführungszeit können nur zwei Situationen eintreten: Entweder referenziert `lastentry` ein Objekt vom Typ `Entry`, sodass garantiert ist, dass das Objekt ein Attribut `element` besitzt, oder `lastentry` enthält die Referenz `null`; in diesem Fall terminiert die Ausführung abrupt mit der Ausnahme `NullPointerException` (vgl. S. 81). Darüber hinaus gewährleistet die Typisierung, dass die angegebene `return`-Anweisung bei normaler Terminierung ein Objekt vom Typ `ListElem` zurückliefert. Besäße Java kein Typkonzept, könnte die Variable `lastentry` beliebige Objekte referenzieren, insbesondere auch Objekte, die kein Attribut `element` besitzen. Um fehlerhafte Speicherzugriffe zu verhindern, müsste deshalb zur Ausführungszeit bei jedem Zugriff geprüft werden, ob das referenzierte Objekt ein Attribut `element` besitzt.

Die Diskussion der Typisierung von Variablen und Attributen lässt sich direkt auf Methoden übertragen. Die Typisierung garantiert, dass das Empfängerobjekt, das bei einem Methodenaufruf ermittelt wurde, eine Methode mit entsprechender Signatur besitzt. Die Typisierung verbessert also die Möglichkeiten, Programme automatisch zur Übersetzungszeit zu überprüfen, und führt im Allgemeinen auch zu einem Effizienzgewinn, da Prüfungen zur Ausführungszeit entfallen können. Außerdem bietet die Typisierung gleichzeitig eine rudimentäre Dokumentation der Programmelemente.

Den Vorteilen, die die Typisierung mit sich bringt, stehen im Wesentlichen zwei Nachteile gegenüber. Erstens wird der Schreibaufwand größer und der Programmtext länger. Zweitens kann die Typisierung auch wie ein Korsett wirken, das die Flexibilität des Programmierers einengt. Um dies zu illustrieren, wenden wir uns nochmals der Klasse `LinkedList` von Abb. 2.9 zu. Würde Java keine korrekte Typisierung erzwingen, könnten wir die Implementierung von `LinkedList` nutzen, um Listenelemente beliebigen Typs zu verarbeiten. Die Typisierung schreibt aber vor, dass nur Objekte vom Typ `ListElem` in die Liste eingefügt werden dürfen. Wollen wir die Listenimplementierung zur Verwaltung von String-Objekten verwenden, müssen wir den gesamten Programmtext kopieren und überall den Typbezeichner `ListElem` durch `String` ersetzen. Derartige Kopier-/Ersetzungs-Operationen blähen die Programme auf und erschweren die Wartung. Gerade im Zusammenhang

mit den häufig vorkommenden Behälterklassen ist ein solches Vorgehen inakzeptabel.

Im Wesentlichen gibt es zwei Ansätze, um die geschilderte Inflexibilität zu überwinden, ohne die Vorteile der Typisierung zu verlieren:

1. Man unterstützt Subtyping (vgl. Abschn. 1.3.2, S. 45). Bei diesem Ansatz kann man Objekte des deklarierten Elementtyps und aller seiner Subtypen in einem Behälter wie `LinkedList` speichern. Sprachen mit Subtyping besitzen i.A. auch einen allgemeinsten Typ, in Java `Object`, der Supertyp aller anderen Typen<sup>12</sup> ist. Verwendet man diesen Typ als Elementtyp, kann man Objekte beliebigen Typs in einer solchen Liste speichern. Subtyping wird in Kap. 3 ausführlich behandelt und war die einzige Antwort, die Java bis zur Version 1.4 auf das skizzierte Problem gab.
2. Man ermöglicht es, Typen bzw. Klassen zu parametrisieren; dieser Ansatz wird im nächsten Absatz erläutert.

Beide Ansätze haben ein unterschiedliches Anwendungsprofil (siehe [Mey00], Kap. 19). Deshalb unterstützen die Sprachen Eiffel, C++ und Ada95 beide Techniken. Ab der Version 5.0 unterstützt auch Java beide Ansätze.

### 2.1.6.1 Parametrisierung von Klassen

Klassen können einen oder mehrere Typparameter besitzen. Solche Klassen nennt man *parametrische Klassen* oder auch *generische Klassen*, da man durch Instanzieren der Parameter unterschiedliche Typen „generieren“ kann.

*parametrische  
Klassen*

Typparameter einer Klassendeklaration werden in Java 5.0 in spitze Klammern eingeschlossen und dem Klassennamen angehängt. Wir folgen der Konvention, Typparameter in Großbuchstaben anzugeben. Ein Typparameter kann im Rumpf der Klassen wie ein normaler Typname verwendet werden. Parametrisiert man die Klassen `Entry` und `LinkedList` bezüglich des Elementtyps, ergeben sich die in Abbildung 2.10 gezeigten Klassendeklarationen.

---

<sup>12</sup>Wertetypen sind in Java keine Subtypen von `Object` und können daher nur über einen Umweg, den wir in Kap. 3 kennen lernen werden, in solchen Behältern verwendet werden.

```

class Entry<ET> {
    ET element;
    Entry<ET> next;
    Entry<ET> previous;

    Entry(ET element, Entry<ET> next, Entry<ET> previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
    }
}

class LinkedList<ET> {
    Entry<ET> header = new Entry<ET>(null, null, null);
    int size = 0;

    /* Constructs an empty Linked List. */
    LinkedList() {
        header.next = header;
        header.previous = header;
    }
    /* Returns the last Element in this List. */
    ET getLast() {
        if( size == 0 ) throw new NoSuchElementException();
        return header.previous.element;
    }
    /* Removes and returns the last Element from this List. */
    ET removeLast() {
        Entry<ET> lastentry = header.previous;
        if(lastentry == header) throw new NoSuchElementException();
        lastentry.previous.next = lastentry.next;
        lastentry.next.previous = lastentry.previous;
        size--;
        return lastentry.element;
    }
    /* Appends the given element to the end of this List. */
    void addLast(ET e) {
        Entry<ET> newEntry = new Entry<ET>(e, header, header.previous);
        header.previous.next = newEntry;
        header.previous = newEntry;
        size++;
    }
    /* Returns the number of elements in this List. */
    int size() {
        return size;
    }
}

```

Abbildung 2.10: Entry und LinkedList als parametrische Klassen

Dieses Beispiel demonstriert bereits,

- wie man parametrische Typen und Typparameter von Klassen bei der Deklaration von Attributen und methodenlokalen Variablen einsetzt (`ET element; Entry<ET> next; ...`),
- wie man sie als Parametertypen verwenden kann (`Entry(ET element, Entry<ET> next, Entry<ET> previous)`) und
- wie man Objekte solcher Typen erzeugen kann (`new Entry<ET>(null, null, null);`).

Grundsätzlich gilt, dass bei der Deklaration von Variablen einschließlich formaler Parameter, die als Typ eine parametrische Klasse verwenden, die aktuellen Typparameter der Klasse (in unserem Beispiel ist es nur einer) in spitzen Klammern hinter dem Klassennamen aufgeführt werden. Dasselbe gilt bei der Instanziierung von parametrischen Klassen. Die Typparameter einer Klasse tauchen in der Klassendeklaration selbst **nicht** mehr hinter dem Konstruktornamen auf (`LinkedList() { ... }`).

Parametrische Klassen kann man bei der Anwendung mit unterschiedlichen Typen instanziiieren. Dabei wird der formale Typparameter, im Beispiel `LinkedList` also `ET`, an allen Stellen konsistent durch einen Typnamen ersetzt. Abbildung 2.11 demonstriert die Instanziierung der parametrischen Listenklasse `LinkedList<ET>` zu den beiden verschiedenen Listenklassen `LinkedList<String>` und `LinkedList<W3Seite>` mit den Elementtypen `String` und `W3Seite`.

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

        LinkedList<String> ls = new LinkedList<String>();
        ls.addLast("erstes Element");
        ls.addLast("letztes Element");
        ls.getLast().indexOf("Elem"); // liefert 8

        LinkedList<W3Seite> lw3 = new LinkedList<W3Seite>();
        lw3.addLast( new W3Seite("Erste Seite","Kein Text") );
        lw3.addLast( new W3Seite("Zweite Seite","Leer") );
        lw3.getLast().indexOf("Elem"); // Programmierfehler,
        // der vom Uebersetzer automatisch entdeckt wird
    }
}
```

Abbildung 2.11: Implementierung eines rudimentären Browsers

Beachtenswert ist jeweils der Aufruf von `getLast`: Auf Grund der Typinformation von `ls` bzw. `lw3` kann der Übersetzer den Typ des Ergebnisses

ermitteln. Im ersten Fall ist dies `String`; der Aufruf von `indexOf` ist also korrekt. Im zweiten Fall erkennt der Übersetzer einen Programmierfehler: Objekte des Typs `W3Seite` besitzen keine Methode `indexOf`.

Dahingegen kann die Klasse `LinkedList` in ihrer Implementierung keine Methoden von `ET` aufrufen, da sie nichts über den ihr als Typparameter übergebenen Typ `ET` weiß. Man könnte sich z.B. vorstellen, die Klasse `LinkedList` um eine Methode `printAll` zu erweitern, die alle in der Liste gespeicherten Elemente ausdrückt. Um diese Aufgabe erfüllen zu können, müßten die in der Liste gespeicherten Elemente beispielsweise alle über eine Methode `drucken` verfügen, die `LinkedList` zum Ausdrucken eines jeden Elements verwenden könnte. Wüßte `LinkedList` z.B., dass `ET` über eine Methode `drucken` verfügt, wäre dieses Problem gelöst. In Java wird dieses Problem mit beschränkten parametrischen Typen angegangen, die wir in Kap. 3 im Kontext von Subtyping kennen lernen werden.

Parametrische Typen werden seit Java 5.0 intensiv im *Collection-Framework* (Paket `java.util`) genutzt. Hier werden diverse Behältertypen definiert für Listen, Mengen etc. Diese Typen verwenden ab der Version 5.0 Typparameter für ihren Elementtyp. Daher kann man jetzt z.B. Listen und Mengen deklarieren, die nur Elemente eines bestimmten Elementtyps enthalten können, ähnlich, wie man Felder mit einem bestimmten Elementtyp festlegen kann. `ArrayList<Integer>` typisiert z.B. eine Liste, die nur Elemente vom Typ `Integer` enthalten kann; `ArrayList<String>` läßt nur Elemente vom Typ `String` zu.

Bisher haben wir nur Beispiele von Klassen mit einem Typparameter gesehen. Das folgende, aus [NW07] entnommene Beispiel zeigt eine Klasse mit zwei Typparametern:

```
public class Pair<T, U> {
    private final T first;
    private final U second;
    public Pair(T first, U second) {
        this.first = first;
        this.second = second;
    }
    public T getFirst() { return first; }
    public U getSecond() { return second; }
}
```

Eine mögliche Variablendeklaration und Instanziierung zeigt das folgende Beispiel:

```
Pair<String, Integer> pair;
pair = new Pair<String, Integer>("one", new Integer(2));
```

### 2.1.6.2 Klassenattribute und -methoden und Überladen von Methodennamen im Kontext parametrischer Typen

Im Kontext parametrischer Klassen gelten einige Besonderheiten für Klassenattribute und -methoden sowie beim Überladen von Methodennamen, die mit der Umsetzung von parametrischen Klassen beim Übersetzen in Bytecode zusammenhängen. Verschiedene Instanzen derselben parametrischen Klasse, also z.B. `LinkedList<Integer>` und `LinkedList<String>`, werden beim Übersetzungsvorgang auf ein- und dieselbe Klasse `LinkedList` abgebildet. Dieser Vorgang wird auch als *Löschung* (engl. *erasure*) bezeichnet.

**Klassenattribute und -methoden** Da sämtliche Instanzen einer parametrischen Klasse durch Löschung auf dieselbe Klasse zurückgeführt werden, verwenden alle Instanzen aller aus einer solchen Klasse durch Einsetzen eines Typparameters generierten Typen dieselben statischen Komponenten.

Klassenattribute und -methoden dürfen daher die Typparameter ihrer Klasse nicht verwenden. Da alle statischen Komponenten einer parametrischen Klasse unabhängig von deren Typparametern sind, darf beim Zugriff auf ein statisches Element der Klassenname nicht mit Typparametern versehen werden.

Die folgenden, [NW07] entnommenen und nur leicht modifizierten Beispiele demonstrieren diese Sachverhalte. Zunächst zeigen wir den Quellcode der Klasse `Cell`, die mit einem Typparameter `T` versehen ist. Jede Instanz eines aus ihr generierten Typs kann einen Wert vom bei der Generierung verwendeten Typ speichern sowie eine ID in Form eines Integer-Werts, der diese Zelle eindeutig identifiziert. Die ID wird fortlaufend vergeben, unabhängig davon, welcher aktuelle Typparameter für `T` eingesetzt wird.

```
class Cell<T> {
    private final int id;
    private final T value;

    // statisches Attribut, aus dem die IDs berechnet werden
    private static int count = 0;

    public Cell(T value) {
        this.value = value;
        id = nextId();
    }
    // statische Methode, die die naechste ID berechnet
    private static int nextId() {
        return count++;
    }
    public static int getCount() {
        return count;
    }
    public T getValue() {
```

```

        return value;
    }
    public int getID() {
        return id;
    }
}

class TestCell {
    // Die main-Methode generiert zwei Zellen mit verschiedenem
    // Typ fuer ihre Werte, gibt deren IDs (0 und 1) aus
    // sowie den Wert des Zaehler count (2).
    public static void main(String[] args) {
        Cell<String> a = new Cell<String>("one");
        Cell<Integer> b = new Cell<Integer>(2);

        System.out.println(a.getID());
        System.out.println(b.getID());
        System.out.println(Cell.getCount());
    }
}

```

Da, wie bereits oben erwähnt, alle Instanzen aller aus einer parametrischen Klasse durch Einsetzen eines Typparameters generierten Typen dieselben statischen Komponenten verwenden, greifen alle Zellen auf dieselbe statische Variable `count` zurück. Daher wird bei der Erzeugung einer String-Zelle und einer Integer-Zelle in obigem Programm dieselbe Variable `count` hochgezählt, sodass beide Zellen fortlaufende IDs (0 und 1) erhalten.

`System.out.println(Cell.getCount());` demonstriert auch die korrekte Verwendung des Klassennamens beim Zugriff auf ein statisches Element der Klasse `Cell<T>`.

`Cell<Integer>.getCount();` würde zu einem Übersetzungsfehler führen. Dem Klassennamen darf beim Aufruf **kein** Typparameter folgen.

Wüßte man nicht um die Einschränkungen, die für statische Komponenten einer parametrischen Klasse gelten, so würde man folgende Variante `Cell2<T>` der Klasse `Cell<T>` für korrekt halten. `Cell2<T>` besitzt ein zusätzliches Attribut `values`, in dem eine Liste aller Werte gehalten wird, die je in einer Zelle gespeichert wurden. Die Methode `getValues` gibt diese Liste zurück.

```

public class Cell2<T> {
    // Attributdeklarationen wie in Cell ...
    private static List<T> values = new ArrayList<T>();

    public Cell2(T value) {
        this.value = value;
        id = nextId();
        values.add(value);
    }
}

```

```

private static int nextId() {...}
public static int getCount() {...}
public int getID() {...}
public T getValue() {...}

public static List<T> getValues() {
    return values;
}
}

```

Der Übersetzer moniert aber die neue Attributdeklaration für `values` ebenso wie die neue Methodendeklaration für `getValues`. Da beide Komponenten statisch sind, dürfen sie den Typparameter `T` nicht verwenden.

**Überladen von Methodennamen** Die Löschung hat auch Auswirkungen auf das Überladen von Methodennamen. Man würde annehmen, dass die folgenden beiden Methodendeklarationen von `allZero` (siehe [NW07]) erlaubt sind, da sie sich offensichtlich im Typ ihres formalen Parameters unterscheiden, der einmal `List<Integer>` ist und einmal `List<String>`. `List` ist dabei ein im Paket `java.util` definierter parametrischer Listen-Typ.

```

public class Overload2 {
    public static boolean allZero(List<Integer> ints) {
        for (int i : ints) if (i != 0) return false;
        return true;
    }

    public static boolean allZero(List<String> strings) {
        for (String s : strings) if (s.length() != 0) return false;
        return true;
    }
    ...
}

```

Der Übersetzer meldet aber einen Fehler, da beide Typen `List<Integer>` und `List<String>` auf denselben Typ `List` abgebildet werden, sodass sich die beiden Methoden in ihrem Parametertyp nicht mehr unterscheiden. Akzeptiert wird dagegen der folgende Programmcode (siehe [NW07]):

```

public class Overload {
    public static int sum(List<Integer> ints) {
        int sum = 0;
        for (int i : ints) sum += i;
        return sum;
    }

    public static String sum(List<String> strings) {

```



```

        StringBuffer sum = new StringBuffer();
        for (String s : strings) sum.append(s);
        return sum.toString();
    }
}

public class OverloadTest {
    public static void main(String[] args) {
        List<Integer> intList = new ArrayList<Integer>(3);
        intList.add(3);
        intList.add(5);
        intList.add(7);
        System.out.println(Overload.sum(intList));

        List<String> stList = new ArrayList<String>(3);
        stList.add("13 ");
        stList.add("15 ");
        stList.add("17 ");
        System.out.println(Overload.sum(stList));
    }
}

```

Hier führt die Löschung auch zum selben Typ des formalen Parameters, nämlich `List`, aber der Compiler kann beide `sum`-Methoden anhand ihres Rückgabetyps unterscheiden, der einmal `int` und einmal `String` ist.

Diese Entscheidung, den Rückgabetypp nur im Kontext parametrischer Typen zur Auflösung der Überladung heranzuziehen, ist kaum nachvollziehbar und selbst die Firma Sun hat dieses Verhalten mittlerweile als einen Bug eingestuft.<sup>13</sup> Es ist davon auszugehen, dass mit Java in Version 7 obiges Beispiel zu einem Compilerfehler führen wird.

## 2.2 Kapselung und Strukturierung von Klassen

Der vorangegangene Abschnitt hat erläutert, wie man mit Klassen Objekte beschreiben kann. Dabei haben wir verschiedene Fragen zunächst zurückgestellt:

- Wie kann man Objekte vor unerwünschtem Zugriff schützen?
- Wie lassen sich Mengen von Klassen strukturieren?
- In welchen Beziehungen können Klassen bzw. ihre Objekte zueinander stehen?

Erste Antworten auf derartige Fragestellungen enthält dieser Abschnitt.

<sup>13</sup>Bug Nr. 6182950, [http://bugs.sun.com/view\\_bug.do?bug\\_id=6182950](http://bugs.sun.com/view_bug.do?bug_id=6182950)

### 2.2.1 Kapselung und Schnittstellenbildung: Erste Schritte

Bisher sind wir stillschweigend davon ausgegangen, dass jeder Benutzer eines Objektes auf alle Attribute zugreifen kann, dass er alle Attribute verändern kann und dass er alle Methoden und Konstruktoren aufrufen kann. Programmtechnisch gesehen heißt das, dass es an jeder Programmstelle erlaubt ist, die genannten Operationen zu nutzen; insbesondere dürfen die Attribute eines Objektes einer Klasse *K* auch in Methoden geändert werden, die nicht zu *K* gehören. Im Wesentlichen gibt es zwei Gründe, eine derart freizügige Benutzung von Objekten zu unterbinden:

1. Der Benutzer einer Klasse kennt die Implementierung selten in allen Details. Er läuft deshalb leicht Gefahr, durch unsachgemäße Benutzung von Objekten Konsistenzkriterien zu verletzen, die für das korrekte Funktionieren der Methoden notwendig sind.
2. Bestimmte Implementierungsaspekte einer Klasse sollten vor den Benutzern verborgen bleiben, sodass Änderungen dieser Aspekte keine Änderungen bei den Anwendungen der Klasse notwendig machen.

Anhand kleiner Beispiele wollen wir diese zwei Gründe etwas näher erläutern. Wir betrachten zunächst die Klasse `LinkedList` von S. 101. Damit ihre Methoden korrekt funktionieren, muss das Attribut `size` den richtigen Wert haben und die Entry-Objekte müssen wie in Abb. 2.8 miteinander verkettet sein: Ist das Attribut `size` falsch besetzt, könnte die Ausnahme `NoSuchElementException` erzeugt werden, obwohl die Liste nicht leer ist. Manipuliert ein Benutzer beispielsweise eine Liste direkt durch Verändern der Attribute (also ohne Methoden zu verwenden) und vergisst dabei, dem `previous`-Attribut des letzten Entry-Objektes die Referenz auf den Header-Entry zuzuweisen, wird die Listenrepräsentation inkonsistent und die Methode `removeLast` wird sich im Allgemeinen nicht mehr korrekt verhalten. Die zu beachtenden Konsistenzkriterien einer Klasse entsprechen den invarianten Eigenschaften aller Objekte dieser Klasse. Deshalb spricht man vielfach auch von den *Klasseninvarianten*.

*Klassen-  
invariante*

Hinter dem zweiten Grund steht das Prinzip des *Information Hiding*, im Deutschen oft als *Geheimnisprinzip* bezeichnet: Man stelle dem Benutzer nur die Schnittstelle zur Verfügung, die er für die Anwendung der Klasse braucht. Alle hinter der Schnittstelle verborgenen Implementierungsteile können dann geändert werden, ohne dass dadurch Änderungen bei den Anwendungen notwendig werden. Natürlich gilt das nur, solange die Schnittstellen von den Änderungen nicht betroffen sind. Wäre beispielsweise garantiert, dass Benutzer der Klasse `W3Seite` niemals auf die Attribute `titel` und `inhalt` zugreifen, könnte man die Implementierung von `W3Seite` wie folgt verändern, ohne dass Benutzer davon informiert werden müssten: Titel und Inhalt werden zu einer Zeichenreihe zusammengesetzt; dabei wird

*Information  
Hiding*

der Titel wie in HTML durch "<TITLE>" und "</TITLE>" geklammert<sup>14</sup>. Die zusammengesetzte Zeichenreihe wird im Attribut `seite` gespeichert. Die Methoden `getTitel` und `getInhalt` führen dann die Zerlegung durch:

```
class W3Seite {
    private String seite;

    W3Seite( String t, String i ) {
        seite = "<TITLE>" + t + "</TITLE>" + i ;
    }
    String getTitel(){
        int trennIndex = seite.indexOf("</TITLE>");
        return new String( seite.toCharArray(), 7, trennIndex-7 );
    }
    String getInhalt(){
        int trennIndex = seite.indexOf("</TITLE>");
        return new String( seite.toCharArray(), trennIndex+8,
                           seite.length() - (trennIndex+8) );
    }
}
```

Die neue Version der Klasse `W3Seite` illustriert am Beispiel des Attributs `seite`, wie Benutzern der Zugriff auf Komponenten einer Klasse verwehrt werden kann: Durch Voranstellen des Zugriffsmodifikators `private` können Attribute, Methoden und Konstruktoren als *privat* vereinbart werden. Private Programmelemente, d.h. als privat deklarierte Attribute, Methoden und Konstruktoren, dürfen nur innerhalb der Klassendeklaration angewendet werden. Dies wird vom Übersetzer geprüft.

Durch Deklaration eines privaten Konstruktors in einer Klasse *K* kann man insbesondere verhindern, dass *K*-Objekte erzeugt werden können, da der private Konstruktor außerhalb von *K* nicht aufgerufen werden kann und der default-Konstruktor (siehe S. 80) nicht mehr bereitsteht. Dies kann für Klassen sinnvoll sein, die nur statische Methoden und Attribute besitzen; ein typisches Beispiel bietet die Klasse `System`.

**Schnittstellenbildung.** Private Attribute, Methoden und Konstruktoren einer Klasse stehen dem Benutzer nicht zur Verfügung und gehören dementsprechend auch nicht zur öffentlichen Schnittstelle der Klasse. Diese explizite Trennung zwischen der öffentlichen Schnittstelle, d.h. der Benutzersicht, und der privaten Schnittstelle, d.h. der Implementierungssicht, bringt die oben angeführten Vorteile mit sich und führt häufig zu einem klareren Klassenentwurf: Der Benutzer kann sich auf das Erlernen der öffentlichen Schnittstelle

<sup>14</sup>Dabei gehen wir davon aus, dass die Zeichenreihe "</TITLE>" nicht als Teilzeichenreihe in Titeln enthalten ist.

konzentrieren; der Implementierer besitzt ein Sprachkonstrukt, um interne Implementierungsteile zu kapseln und klar von extern verfügbaren Teilen zu trennen. In vielen Fällen erweist es sich dabei als sinnvoll, Attribute als privat zu deklarieren und den Zugriff nur über Methoden zu ermöglichen. Die Methoden sollten dann so entworfen werden, dass klar zwischen Methoden unterschieden werden kann, die nur lesend auf Attribute zugreifen, und solchen, die den Zustand von Objekten verändern. (Beispielsweise greifen die Methoden `getTitel` und `getInhalt` der Klasse `W3Seite` nur lesend auf das Attribut `seite` zu.)



In den folgenden Abschnitten und in Kap. 3 werden wir weitere Sprachkonstrukte zur Kapselung und zur Beschreibung von Schnittstellen kennen lernen.

## 2.2.2 Strukturieren von Klassen

Bisher haben wir die Klassendeklarationen eines Programms als eine unstrukturierte Menge betrachtet. Es gab weder die Möglichkeit, eine Klasse einer anderen im Sinne einer Blockschachtelung unterzuordnen, noch die Möglichkeit mehrere Klassen zu einer Einheit zusammenzufassen. Java stellt beide Strukturierungsmöglichkeiten zur Verfügung: Eine Klasse, die innerhalb einer anderen deklariert ist, wird in Java eine innere Klasse genannt; Unterstützung bei der Gruppierung von Klassen zu größeren Einheiten bieten die sogenannten Pakete. Hand in Hand mit der Behandlung dieser Strukturierungskonstrukte werden wir die dazugehörigen Kapselungstechniken erläutern.

### 2.2.2.1 Innere Klassen

Die Verwendung von Blockschachtelung in Programmiersprachen hat zwei Gründe:

1. Sie strukturiert die Programme.
2. Sie legt Gültigkeitsbereiche für Namen fest.

Die Technik der Schachtelung von Blöcken wurde in Java auf Klassen übertragen: Klassen können als Komponenten anderer Klassen oder innerhalb von Blöcken deklariert werden (Beispiele der zweiten Form werden wir in Kap. 3 kennen lernen). Solche Klassen nennt man *innere Klassen*. Innere Klassen bieten u.a. die Möglichkeit, kleinere Hilfsklassen nahe bei den Programmstellen zu deklarieren, an denen sie gebraucht werden. In Kombination mit den Kapselungstechniken, die wir in Abschnitt 2.2.1 behandelt haben, ergibt sich darüber hinaus ein sehr flexibler Mechanismus, um die Zugreifbarkeit von Implementierungsteilen zu regeln.

*innere Klasse*

Innere Klassen sind ein recht komplexes Sprachkonstrukt. In diesem Abschnitt werden wir uns auf die Anwendung innerer Klassen zu Strukturierungs- und Kapselungszwecken konzentrieren. Andere Anwendungen werden wir in späteren Kapiteln behandeln. Zunächst betrachten wir sogenannte statische innere Klassen am Beispiel von `LinkedList`. Der zweite Absatz stellt nicht-statische innere Klassen am Beispiel von Iteratoren vor.

**Statische innere Klassen.** In vielen Fällen kommt es vor, dass man zur Implementierung einer Klasse  $K$  Hilfsklassen benötigt, d.h. Klassen, die der Benutzer von  $K$  nicht kennen muss und die besser vor ihm verborgen bleiben. Eine solche Hilfsklasse ist die Klasse `Entry` für die Implementierung doppeltverketteter Listen (vgl. Abb. 2.9, S. 101). Indem wir die Klasse `Entry` zu einer privaten inneren Klasse von `LinkedList` machen und außerdem die Attribute von `LinkedList` als privat deklarieren, können wir die gesamte Implementierung von doppeltverketteten Listen kapseln. Damit ist kein Benutzer mehr in der Lage, die Invarianten der Klasse zu verletzen. Insgesamt erhält die Klasse `LinkedList` damit folgendes Aussehen (Methodenrumpfe bleiben wie in Abb. 2.9):

```
class LinkedList {
    private Entry header = new Entry(null,null,null);
    private int size = 0;

    LinkedList() {
        header.next = header;
        header.previous = header;
    }
    ListElem getLast() { ... }
    ListElem removeLast() { ... }
    void addLast(ListElem e) { ... }
    int size() { ... }

    private static class Entry {
        private ListElem element;
        private Entry next;
        private Entry previous;

        Entry(ListElem element,Entry next,Entry previous){ ... }
    }
}
```

Die innere Klasse `Entry` ist als privat deklariert; d.h. ihr Typname ist außerhalb der Klasse `LinkedList` nicht ansprechbar (ein Beispiel für eine nicht private innere Klasse werden wir gleich behandeln). Allgemeiner gilt, dass private Programmelemente überall in der am weitesten außen liegenden

```

class MyClassIsMyCastle {
    private static int streetno = 169;

    private static class FirstFloor {
        private static class DiningRoom {
            private static int size = 36;
            private static void mymessage(){
                System.out.print("I can access streetno");
                System.out.println(": "+ streetno );
            }
        }
    }
    private static class SecondFloor {
        private static class BathRoom {
            private static int size = 16;
            private static void mymess(){
                System.out.print("I can access the ");
                System.out.print("dining room size: ");
                System.out.println(""+ FirstFloor.DiningRoom.size);
            }
        }
    }
    public static void main( String[] args ) {
        FirstFloor.DiningRoom.mymessage();
        SecondFloor.BathRoom.mymess();
    }
}

```

Abbildung 2.12: Schachtelung und Zugriff bei inneren Klassen

Klasse zugreifbar sind, aber nicht außerhalb davon; insbesondere sind private Programmelemente in weiter innen liegenden Klassen zugreifbar.

Dazu betrachten wir zwei Beispiele. Wie im obigen Programmfragment von `LinkedList` demonstriert, sind die privaten Attribute von `Entry` im Konstruktor `LinkedList` zugreifbar, d.h. außerhalb der Klassendeklaration von `Entry`. Ein komplexeres Szenario illustrieren die Klassen in Abb. 2.12. Darüber hinaus zeigen sie, wie man mit zusammengesetzten Namen der Form *InKl.ProgEl* Programmelemente *ProgEl* in inneren Klassen *InKl* ansprechen kann. Beispielsweise bezeichnet `FirstFloor.DiningRoom` die innere Klasse `DiningRoom` in Klasse `FirstFloor`. Deren Attribut `size` lässt sich mittels `FirstFloor.DiningRoom.size` ansprechen.

Die Klassen in Abb. 2.12 demonstrieren drei verschiedene Zugriffsverhalten:

1. Umfassende Klassen können auf private Programmelemente in tiefer geschachtelten Klassen zugreifen: In der Methode `main` wird auf die Methoden der inneren Klassen `DiningRoom` und `BathRoom` zugegriffen.

2. Innere Klassen können auf private Programmelemente umfassender Klassen zugreifen: In der Methode `mymessage` wird auf das private Attribut `streetno` der am weitesten außen liegenden Klasse zugegriffen.
3. Innere Klassen können auf private Programmelemente anderer inneren Klassen zugreifen, wenn es eine beide umfassende Klasse gibt: In der Methode `mymess` wird auf das private Attribut `size` der Klasse `DiningRoom` zugegriffen.

Alle bisher gezeigten inneren Klassen sind mittels des Schlüsselworts `static` als statisch deklariert. Ähnlich wie bei statischen Attributen bzw. Methoden ist eine statische innere Klasse nur der umfassenden Klassendeklaration zugeordnet, aber nicht deren Instanzen. Insbesondere darf sie nur statische Attribute oder Methoden der umfassenden Klasse benutzen. Beispielsweise wäre es unzulässig, innerhalb der Konstruktordeklaration von `Entry` auf das Attribut `header` zuzugreifen. Statische innere Klassen dienen dazu, ansonsten gleichrangige Klassen zu strukturieren (dies war der erste Grund zur Verwendung von Schachtelung; siehe oben) und die Kapselungsmöglichkeiten zu verfeinern.

**Nicht-statische innere Klassen.** Nicht-statische innere Klassen bieten einen mächtigeren Mechanismus als statische innere Klassen: Sei *IN* eine nicht-statische innere Klasse einer Klasse *K*; dann bekommt jedes *IN*-Objekt implizit eine Referenz auf ein zugehöriges *K*-Objekt. Das zugehörige *K*-Objekt kann in der inneren Klasse über den zusammengesetzten Bezeichner `K.this` angesprochen werden. Dadurch kann das Objekt der inneren Klasse insbesondere auf die Attribute und Methoden des zugehörigen *K*-Objekts zugreifen; sofern es keine Namenskonflikte mit den Attributen der inneren Klasse gibt, kann dafür direkt der Attribut- bzw. Methodenname verwendet werden (also z.B. statt `K.this.attr` direkt `attr`).

Durch diese Verbindung zu einem Objekt der umfassenden Klasse und die Möglichkeit, auf dessen Attribute und Methoden zugreifen zu können, insbesondere auch auf die privaten, lassen sich auch komplexe Kapselungsanforderungen meistern. Eine detaillierte Erörterung der resultierenden Möglichkeiten würde in diesem Zusammenhang aber zu weit führen. Wir begnügen uns hier mit der Diskussion eines typischen Beispiels: der Realisierung von Iteratoren für unsere Listenklasse. Iteratoren stellen eine wichtige Programmiertechnik dar, die jeder Programmentwickler kennen sollte. (Weitere Beispiele nicht-statischer innerer Klassen werden wir in den folgenden Kapiteln kennen lernen.)

Beim Arbeiten mit Behältern (Mengen, Listen, Schlangen, etc.) steht man häufig vor der Aufgabe, effizient über alle Elemente im Behälter zu iterieren, beispielsweise um auf jedem Element eine bestimmte Operation auszuführen

oder um ein bestimmtes Element zu suchen. Diese allgemeine Aufgabenstellung wollen wir anhand eines konkreten Beispiels genauer betrachten und zwar anhand einer einfachen Implementierung der Klasse `W3Server` aus Abschn. 2.1.3, S. 85.

Bei einem W3-Server kann man W3-Seiten unter einer Adresse ablegen und holen. Die Adresse ist eine beliebige Zeichenreihe. Die Seiten werden in einer Liste verwaltet, deren Listenelemente Paare sind, die aus einer Adresse und einer W3-Seite bestehen:

```
class ListElem {
    private String adr;
    private W3Seite seite;

    ListElem( String a, W3Seite s ){
        adr    = a;
        seite  = s;
    }
    String  getAdr  () { return adr; }
    W3Seite getSeite() { return seite; }
    void    setSeite( W3Seite s ) { seite = s; }
}
```

Die Liste solcher Paare wird beim W3-Server im Attribut `adrSeitenListe` gespeichert. Wird versucht, unter einer Adresse, zu der keine Seite vorhanden ist, eine Seite zu holen, soll eine Fehlerseite zurückgeliefert werden:

```
class W3Server {
    LinkedList adrSeitenListe      = new LinkedList();
    static final W3Seite fehlerseite =
        new W3Seite("Fehler", "Seite nicht gefunden");
    void ablegenSeite( W3Seite s, String sadr ) { ... }
    W3Seite holenSeite( String sadr ) { ... }
}
```

Wir wollen hier die Methode `holenSeite` entwickeln, wobei die Version der Klasse `LinkedList` von S. 114 verwendet werden soll. Um es uns einfach zu machen, verwenden wir lineare Suche: Die Adressen-Seiten-Liste wird schrittweise durchlaufen; dabei wird jeweils die Adresse der aktuellen Seite mit dem Parameter verglichen; wird eine Seite gefunden, wird sie zurückgegeben; ansonsten wird schließlich die Fehlerseite abgeliefert:

```
W3Seite holenSeite( String sadr ) {
    while( adrSeitenListe hat nächstes Element ) {
        ListElem adp = nächstes Element ;
        if( adp.getAdr().equals( sadr ) ) return adp.getSeite();
    }
    return fehlerseite;
}
```



Wir brauchen nur noch die informellen Anweisungen durch geeigneten Programmcode zu ersetzen. Aber wie soll das gehen? Die Methoden der Klasse `LinkedList` erlauben uns zwar, Listen auf- und abzubauen und festzustellen, ob eine Liste leer ist; das Iterieren über alle Elemente wird aber nicht unterstützt. Und außerhalb der Klasse `LinkedList` können wir das auch nicht realisieren, da die Kapselung den Zugriff auf die `Entry`-Objekte verbietet.

Zur Lösung dieses Problems versehen wir die Klasse `LinkedList` mit Iteratoren: Ein Listen-Iterator ist ein Objekt, das eine Position in einer Liste repräsentiert. Die Position kann sich zwischen zwei Listenelementen, am Anfang oder am Ende der Liste befinden. Für die Beschreibung der Funktionalität von Iteratoren gehen wir davon aus, dass die Listenelemente wie in Abb. 2.8, S. 100, von links nach rechts geordnet sind und dass die Elemente von links nach rechts aufsteigend mit Indizes beginnend bei null durchnummeriert sind. Die Position wird in einem Attribut `nextIndex` vom Typ `int` gespeichert, wobei `nextIndex` immer den Index des Elements rechts von der Position enthält; in der Anfangsposition ist also `nextIndex = 0`. Außerdem besitzt jedes Iterator-Objekt ein Attribut `next` vom Typ `Entry`, das immer auf das Listenelement mit Index `nextIndex` verweist. Ein Iterator besitzt die Methoden `hasNext` zur Prüfung, ob die Position am Ende der Liste ist, und `next` zum Zugriff auf das nächste Element und Weiterschalten der Position. Damit Iteratoren auf die `Entry`-Objekte zugreifen können, deklarieren wir die Klasse `ListIterator` in Abbildung 2.13 als innere Klasse von `LinkedList`.

Die Klasse `ListIterator` ist keine statische Klasse. Damit hat sie Zugriff auf die Attribute der umfassenden Klasse, im Beispiel auf die Attribute `header` und `size` der Klasse `LinkedList`. Um solche Zugriffe zur Ausführungszeit zu ermöglichen, besitzt jedes Objekt einer (nicht-statischen) inneren Klasse eine Referenz auf ein Objekt der umfassenden Klasse. Diese Referenz wird in einem implizit deklarierten Attribut gespeichert. Jedes `ListIterator`-Objekt besitzt also eine implizite Referenz auf das zugehörige `LinkedList`-Objekt. Diese implizite Referenz wird bei der Erzeugung des Objekts der inneren Klasse hergestellt. In unserem Beispiel bekommt dazu der default-Konstruktor `ListIterator` immer implizit das aktuelle `this`-Objekt seiner Aufrufstelle als Argument mitgeliefert. Die Methode `listIterator` liefert also einen Iterator für das `LinkedList`-Objekt, für das sie aufgerufen wurde.

Damit der Programmierer nicht immer eine Methode wie `listIterator` schreiben muss, um Konstruktoren nicht-statischer innerer Klassen aufrufen zu können, wird von Java auch eine spezielle syntaktische Variante dafür angeboten, bei der dem `new`-Ausdruck wie bei einem Methodenaufruf ein implizites Objekt mitgegeben wird. Bezeichnet beispielsweise `meineListe` eine initialisierte Variable vom Typ `LinkedList`, dann ist der Ausdruck `meineListe.new ListIterator()` äquivalent zu dem Ausdruck `meineListe.listIterator()`.

```
class LinkedList {
    private Entry header = new Entry(null, null, null);
    private int size = 0;
    ... // wie oben

    private static class Entry { ... } // wie oben

    class ListIterator {
        private int nextIndex = 0;
        private Entry next = header.next;

        boolean hasNext() {
            return nextIndex != size;
        }
        ListElem next() {
            if( nextIndex==size )
                throw new NoSuchElementException();
            ListElem elem = next.element;
            next = next.next;
            nextIndex++;
            return elem;
        }
    }

    ListIterator listIterator(){ return new ListIterator(); }
}
```

Abbildung 2.13: Die Klasse `LinkedList` mit einem Listen-Iterator

Die Programmierung und das Verstehen der Implementierung innerer Klassen ist zum Teil recht komplex (insbesondere wenn zusätzlich weitere Sprachmittel verwendet werden; vgl. dazu die vollständige Version der Klasse `LinkedList` im Bibliothekspaket `java.util`). Am Beispiel der Listeniteratoren werden wir aber sehen, dass die Anwendung trotzdem sehr einfach sein kann. Man vergleiche dazu die obige unvollständige Version der Methode `holenSeite` mit der vollständigen, die hier nochmals in ihrem Klassenkontext gezeigt ist:

```
class W3Server {
    LinkedList adrSeitenListe = new LinkedList();
    ...
    W3Seite holenSeite( String sadr ) {
        LinkedList.ListIterator lit =
            adrSeitenListe.listIterator();
        while( lit.hasNext() ) {
            ListElem adp = lit.next();
            if( adp.getAdr().equals(sadr) ) return adp.getSeite();
        }
        return fehlerseite;
    }
}
```

Man beachte das Erzeugen eines Iterators für die Adressen-Seiten-Liste mittels der Methode `listIterator`. Im Übrigen sei nochmals auf die Verwendung der Punkt-Notation bei `LinkedList.ListIterator` hingewiesen; auf diese Weise kann der Typname `ListIterator` außerhalb der Klassendeklaration von `LinkedList` benutzt werden. Einen ähnlichen Mechanismus werden wir im Zusammenhang mit Paketen kennen lernen.

**Parametrische Klassen mit inneren Klassen** Die Sichtbarkeit der Typparameter einer parametrischen Klasse innerhalb ihrer inneren Klassen hängt davon ab, ob es sich um statische oder nicht statische innere Klassen handelt. Eine nicht statische innere Klasse kann in ihrer Implementierung auf die Typparameter ihrer umfassenden Klasse ohne Probleme zugreifen. Bei statischen inneren Klassen ist dies nicht möglich. Soll eine statische innere Klasse auf einen Typparameter ihrer umfassenden Klasse zugreifen, muss man die innere Klasse selbst mit einem Typparameter versehen. Diesem Typparameter der inneren Klasse muss dann bei Variablendeklarationen, Instanziierungen etc. derselbe Typparameter mitgegeben werden, den auch die umfassende Klasse übergeben bekommt. Ein Beispiel für beide Varianten finden wir in der folgenden Deklaration der mit einem Elementtyp `ET` parametrisierten Klasse `LinkedList`, die die parametrische Variante der Implementierung aus Abbildung 2.13 ist. Diese Klasse enthält die Klasse `Entry` als statische innere Klasse und die Klasse `ListIterator` als nicht statische innere Klasse.

Entry muss daher mit einem Typparameter T versehen werden, der bei Verwendung von Entry innerhalb von LinkedList und deren anderen inneren Klassen immer mit ET instanziiert werden muss. ListIterator kommt dagegen ohne eigenen Typparamter aus. Als nicht statische innere Klasse ist ET innerhalb von ListIterator bekannt.

```
class LinkedList<ET> {

    static class Entry<T> {
        T element;
        Entry<T> next;
        Entry<T> previous;

        Entry(T element, Entry<T> next, Entry<T> previous) {
            this.element = element;
            this.next = next;
            this.previous = previous;
        }
    }

    Entry<ET> header = new Entry<ET>(null, null, null);
    int size = 0;

    // Constructs an empty Linked List.
    LinkedList() {
        header.next = header;
        header.previous = header;
    }

    // Returns the last Element in this List.
    ET getLast() {
        if( size == 0 ) throw new NoSuchElementException();
        return header.previous.element;
    }

    // Removes and returns the last Element from this List.
    ET removeLast() {
        Entry<ET> lastentry = header.previous;
        if(lastentry == header) throw new NoSuchElementException();
        lastentry.previous.next = lastentry.next;
        lastentry.next.previous = lastentry.previous;
        size--;
        return lastentry.element;
    }

    // Appends the given element to the end of this List.
    void addLast(ET e) {
        Entry<ET> newEntry = new Entry<ET>(e, header, header.previous);
        header.previous.next = newEntry;
        header.previous = newEntry;
        size++;
    }
}
```

```

// Returns the number of elements in this List.
int size() {
    return size;
}

class ListIterator {
    private int nextIndex = 0;
    private Entry<ET> next = header.next;

    boolean hasNext() {
        return nextIndex != size;
    }
    ET next() {
        if (nextIndex == size)
            throw new NoSuchElementException();
        ET elem = next.element;
        next = next.next;
        nextIndex++;
        return elem;
    }
}

ListIterator listIterator() {
    return new ListIterator();
}

}

public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> ls = new LinkedList<String>();
        ls.addLast("erstes Element");
        ls.addLast("letztes Element");
        ls.getLast().indexOf("Elem"); // liefert 8
        LinkedList<String>.ListIterator it = ls.listIterator();
        while (it.hasNext()) {
            String st = it.next();
            System.out.println(st);
        }

        LinkedList<Integer> li = new LinkedList<Integer>();
        li.addLast( new Integer(1) );
        li.addLast( new Integer(2) );
        li.addLast( new Integer(3) );

        LinkedList<Integer>.ListIterator lit = li.listIterator();
        while (lit.hasNext()) {
            Integer i = lit.next();
            System.out.println(i);
        }

        LinkedList.Entry<Integer> entry =

```

```
        new LinkedList.Entry<Integer>(new Integer(5), null, null);  
    }  
}
```

Die Klasse `LinkedListTest` zeigt die Verwendung von `LinkedList` mit verschiedenen aktuellen Typparametern sowie das Iterieren über die deklarierten Listen mit Hilfe eines Iterators. Das Beispiel zeigt auch, wie man nicht private statische und nicht statische innere Klassen über ihre umfassende parametrische Klasse referenzieren kann. Auf nicht statische innere Klassen kann man zugreifen über den Klassennamen der umfassenden Klasse gefolgt von dem aktuellen Typparameter der Klasse, dem der Name der inneren Klasse durch einen Punkt getrennt folgt: `LinkedList<String>.ListIterator`. Der Zugriff auf eine statische innere Klasse erfolgt durch Angabe des Klassennamens der umfassenden Klasse ohne Typparameter gefolgt vom Namen der inneren Klasse inklusive Typparameter: `LinkedList.Entry<Integer>`.

### 2.2.2.2 Strukturierung von Programmen: Pakete

Größere Programme und vor allem Programmbibliotheken bestehen oft aus einer Vielzahl von Klassen. Es ist von großer Bedeutung für die Softwareentwicklung, diese Klassen geeignet zu organisieren. Dabei sind die folgenden Aspekte zu beachten:

1. Auffinden, Verstehen: Die gute Strukturierung der Klassen eines großen Programms erleichtert es, den Zusammenhang zwischen Softwareentwurf und Programmcode herzustellen und damit das Programm zu verstehen. Dem Benutzer einer Bibliothek sollte es leicht gemacht werden, Klassen in der Bibliothek ausfindig zu machen, die ihm bei seiner Implementierung helfen können.
2. Pflege: Die Strukturierung der Klassen sollte die Abhängigkeiten zwischen den Klassen widerspiegeln können, um Wartung und Pflege der Programme zu verbessern.
3. Zugriffsrechte, getrennte Namensräume: Klassen sollten zu Modulen zusammengefasst werden können, sodass man den Zugriff von einem Modul auf ein anderes einschränken kann und in verschiedenen Modulen getrennte Namensräume hat.
4. Getrennte Übersetzung: Im Zusammenhang mit der Strukturierung muss auch festgelegt werden, welche Programmteile unabhängig voneinander übersetzt werden können.

Über-  
setzungseinheit

Java benutzt sogenannte *Pakete* zur Modularisierung von Programmen. Pakete besitzen eindeutige Namen, werden aber nicht explizit deklariert. Ein Paket umfasst eine Menge sogenannter *Übersetzungseinheiten*. Wie die Bezeichnung suggeriert, lässt sich eine Übersetzungseinheit getrennt von anderen Übersetzungseinheiten übersetzen. Syntaktisch hat eine Übersetzungseinheit in Java die folgende Form:

Paket

```
package Paketname ;
Liste von import-Anweisungen
Liste von Typdeklarationen
```

Beispiele für Übersetzungseinheiten finden Sie in Abbildung 2.14. Der Paketname gibt an, zu welchem Paket die Übersetzungseinheit gehört. Die import-Anweisungen ermöglichen es, Klassen aus anderen Paketen in der Übersetzungseinheit direkt sichtbar zu machen. Die Typdeklarationen bilden den eigentlichen Inhalt einer Übersetzungseinheit. Eine *Typdeklaration* ist entweder eine Klassen- oder eine Schnittstellendeklaration (die Deklaration von Schnittstellen behandeln wir in Kap. 3). Wir gehen im Folgenden davon aus, dass jede Übersetzungseinheit genau einer Datei des benutzten Rechners entspricht. Alle Übersetzungseinheiten eines Pakets müssen sich in einem Dateiverzeichnis befinden. Dabei muss der Name des Dateiverzeichnisses dem Paketnamen entsprechen (was das genau heißt, wird weiter unten erläutert). Demnach können wir jedes Paket mit dem Dateiverzeichnis identifizieren, in dem sich seine Übersetzungseinheiten befinden.

Im Folgenden werden wir anhand unseres Browser-Beispiels zeigen, wie Übersetzungseinheiten aussehen, welche Möglichkeiten der Kapselung es bei Paketen gibt, wie auf Bestandteile anderer Pakete zugegriffen werden kann und wie Pakete hierarchisch strukturiert werden können.

Die acht behandelten Klassen des Beispiels teilen wir auf zwei Pakete auf: das Paket `browse` und das Paket `browse.util`. Das Paket `browse.util` enthält die Klassen, die allgemeine Hilfsaufgaben erledigen, nämlich `Konsole` und `LinkedList`; die restlichen Klassen bilden das Paket `browse`. Wie üblich in Java, sehen wir für jede Klasse eine eigene Übersetzungseinheit, sprich Datei, vor. Die Dateien für das Paket `browse` liegen alle in einem Dateiverzeichnis `browse` mit einem absoluten Namen `PFAD/browse/`, wobei `PFAD` für ein Dateiverzeichnis auf dem benutzten Rechner steht<sup>15</sup>. Die Dateien für das Paket `browse.util` stehen im Dateiverzeichnis `util` mit absolutem Namen `PFAD/browse/util/`. Abbildung 2.14 zeigt den Zusammenhang zwischen Paketen und Dateiverzeichnissen sowie zwischen Übersetzungseinheiten, Klassen und Dateien.

Die durchgezogenen Linien rahmen Übersetzungseinheiten ein und entsprechen damit jeweils einer Datei. Die fetten gestrichelten Linien bezeichnen Paketgrenzen. Die dünnen unterbrochenen Linien markieren die Bereiche von Dateiverzeichnissen. Die Abbildung zeigt insbesondere den Zusam-

<sup>15</sup>Wir benutzen hier, wie unter Unix üblich, den Schrägstrich „/“ als Trenner in Dateipfaden; bei Verwendung anderer Betriebssysteme ist die Pfadsyntax geeignet anzupassen.

Dateiverzeichnis: PFAD/browse/


*Paket*

Dateiverzeichnis

Datei

Abbildung 2.14: Strukturierung von Klassen in Pakete



menhang zwischen der Deklaration des Paketnamens am Anfang einer Übersetzungseinheit und dem Namen des entsprechenden Dateiverzeichnisses.

Zum Arbeiten mit kleineren Programmen gestattet es Java, die Deklaration des Paketnamens in der Übersetzungseinheit wegzulassen. Dann werden die Typendeklarationen der Einheit einem unbenannten Paket zugeordnet. Von diesem Sonderfall haben wir bisher Gebrauch gemacht.



Java nutzt die Baumstruktur von Dateisystemen, um Pakete hierarchisch *anzuordnen*. Die Pakete selber sind aber nicht strukturiert. Insbesondere ist das Paket `browse.util` nicht Teil des Pakets `browse`; beide Pakete sind unabhängig voneinander und die gegenseitige Benutzung muss auf die gleiche Weise geregelt werden, wie dies bei Paketen erforderlich ist, die in getrennten Dateiverzeichnissen stehen.

**Benutzung von Paketen.** Grundsätzlich kann jede Klasse alle Klassen in anderen Java-Paketen dieser Welt benutzen. Es gibt allerdings drei Einschränkungen:

1. Kapselung auf der Ebene von Paketen: Java ermöglicht es, Klassen innerhalb von Paketen zu kapseln und damit dem Zugriff durch andere Pakete zu entziehen (siehe nächsten Absatz).
2. Erreichbarkeit: Das gewünschte Paket/Dateiverzeichnis muss dem benutzten Rechner bzw. dem an ihm arbeitenden Benutzer unter einem geeigneten Namen zugänglich sein.
3. Eindeutigkeit: Die vollständigen Klassennamen bestehend aus Paket- und Klassennamen müssen eindeutig sein.

Auf eine Klasse  $K$  eines fremden Pakets  $q$  kann man zugreifen, indem man den Paketnamen dem Klassennamen voranstellt, d.h. den sogenannten *vollständigen* Namen  $q.K$  verwendet.

*vollständiger  
Name*

Bei der Auflösung der vollständigen Namen kann es zu Mehrdeutigkeiten kommen. Zwei mögliche Ursachen von solchen Namenskonflikten werden im Folgenden aufgezeigt.

Nehmen wir beispielsweise an, dass auf unserem Rechner die Pakete `testpaket` und `nocheinpaket.namen` existieren. Dann können wir in der Klasse `testpaket.Main` die Klasse `Konflikt` des Pakets `nocheinpaket.namen` und damit ihre statische Methode `slogan` wie folgt benutzen:

```
package testpaket;

public class Main {
    public static void main(String[] args) {
        nocheinpaket.namen.Konflikt.slogan();
    }
}
```

wobei die Klasse `Konflikt` beispielsweise wie folgt aussehen könnte:

```
package nocheinpaket.namen;

public class Konflikt {
    public static void slogan() {
        System.out.println("Ein Traum von Namensraum");
    }
}
```

Damit Java-Übersetzer und Java-Laufzeitumgebungen Pakete und Klassen auf dem lokalen oder einem entfernten Rechner finden können, muss man ihnen mitteilen, in welchen Dateiverzeichnissen sie danach suchen sollen. Diese Information erhalten sie üblicherweise über die Umgebungsvariable `CLASSPATH` oder einen ähnlichen Mechanismus, der angibt, wie auf die Pakete bzw. auf die Klassen zugegriffen werden soll. Der `CLASSPATH`-Mechanismus, den wir im Rest des Kurses zugrunde legen, funktioniert auf folgende Weise: Enthält die Umgebungsvariable `CLASSPATH` beispielsweise bei der Übersetzung die Namen der Verzeichnisse `/home/gosling/myjava/` und `/jdk1.5.0_10/classes/`, würde ein Übersetzer die Klasse `Konflikt` in den Dateiverzeichnissen `/home/gosling/myjava/nocheinpaket/namen/` und `/jdk1.5.0_10/classes/nocheinpaket/namen/` suchen. Gäbe es die Datei `Konflikt.class` in beiden Unterverzeichnissen, wäre zunächst nicht klar, welche der beiden Klassen verwendet werden soll. Der Konflikt wird dadurch aufgelöst, dass der Übersetzer die Reihenfolge der in der `CLASSPATH`-Variablen angegebenen Verzeichnisse beachtet. Weiter vorne angegebene Pfade werden zuerst berücksichtigt. Entsprechendes gilt für Laufzeitumgebungen.

Um solche Mehrdeutigkeiten gar nicht erst entstehen zu lassen, sollte man darauf achten, dass Pakete in unterschiedlichen Verzeichnissen verschiedene Namen besitzen.

Eine andere Quelle von Namenskonflikten hängt mit der Punktnotation bei der Benutzung innerer Klassen zusammen. Beispielsweise würde die Existenz der folgenden Klasse dazu führen, dass der Aufruf von `slogan` in der obigen Klasse `Main` mehrdeutig wird:

```
package nocheinpaket;

public class namen {
    public static class Konflikt {
        public static void slogan() {
            System.out.println("Pakete zollfrei importieren");
        }
    }
}
```

Der Übersetzer und das Laufzeitsystem lösen einen solchen Konflikt dadurch auf, dass Klassen vorrangig vor Paketen berücksichtigt werden. In unserem Fall würde daher `nocheinpaket.namen` abgebildet auf die Datei `namen.class` im Unterverzeichnis `nocheinpaket` und nicht auf das Unterverzeichnis `nocheinpaket/namen`.

Diese Art von Mehrdeutigkeit kann verhindert werden, wenn man sich an die Konvention hält, Paketnamen mit Klein- und Klassennamen mit Großbuchstaben zu beginnen.

Da es in der Praxis umständlich und der Lesbarkeit abträglich wäre, Programmelementen aus anderen Paketen immer den Paketnamen voranstellen zu müssen, unterstützt Java einen `import`-Mechanismus. Nach der Deklaration des Paketnamens am Anfang einer Übersetzungseinheit kann, wie auf Seite 124 bereits erwähnt, eine Liste von `import`-Anweisungen angegeben werden, wobei es zwei Varianten von `import`-Anweisungen gibt:

```
import Paketname.Typname ;  
import Paketname.* ;
```

Mit der ersten Deklaration importiert man die Typdeklaration mit Namen *Typname* aus Paket *Paketname*. Der Typ, also insbesondere ein Klassentyp, kann dann in der Übersetzungseinheit direkt mit seinem Namen angesprochen werden, ohne dass der Paketname vorangestellt werden muss. Der importierte Typ muss allerdings als öffentlich deklariert sein (der folgende Absatz „Kapselung auf Paketebene“ beschreibt den Unterschied zwischen öffentlichen und nicht öffentlichen Typen). Mit der zweiten Deklaration importiert man alle öffentlichen Typen des Pakets *Paketname*. Beispielsweise werden durch die Deklaration

```
import java.lang.reflect.*;
```

alle öffentlichen Typen des Pakets `java.lang.reflect` importiert, also insbesondere der Typ `Method`. Bei dieser zweiten Form von `import`-Anweisung werden nur die Typen importiert, für die es ansonsten keine Deklaration in der Übersetzungseinheit gibt. Enthielte eine Übersetzungseinheit mit obiger `import`-Anweisung also eine eigene Typdeklaration `Method`, würde diese in der Übersetzungseinheit verwendet und nicht diejenige aus dem Paket `java.lang.reflect`.

Das Standard-Paket `java.lang` der Java-Bibliothek wird automatisch von jeder Übersetzungseinheit importiert. Es sei nochmals darauf hingewiesen, dass zu diesem Paket nur die Typen gehören, die direkt im `java.lang` entsprechenden Verzeichnis stehen. Insbesondere werden also die Typen des Pakets `java.lang.reflect` nicht automatisch in jede Übersetzungseinheit importiert.

**Kapselung auf Paketebene.** In Abschn. 2.2.1 wurde erläutert, warum man Teile einer Klassenimplementierung dem Zugriff der Benutzer entziehen möchte und wie Java dieses Kapselungskonzept unterstützt. Aus den gleichen Gründen ist es sinnvoll, nur ausgewählte Klassen und nur Teile dieser Klassen der Benutzung in anderen Paketen zugänglich zu machen, d.h. Teile von Paketen zu kapseln.

Programmelemente, d.h. Klassen, Attribute, Methoden und Konstruktoren, deren Benutzung in allen Paketen gestattet sein soll, müssen als *öffentlich* deklariert werden. Dazu stellt man ihrer Deklaration den Modifikator `public` voran. Ein kleines Beispiel bietet die obige Deklaration der öffentlichen Klasse `Konflikt` mit der öffentlichen Methode `slogan`. Programmelemente, die weder öffentlich noch privat sind, können überall in dem Paket benutzt werden, in dem sie deklariert sind; ohne weitere Angabe ist ein Zugriff von außerhalb des Pakets nicht zulässig<sup>16</sup>. Wir sagen, dass solche Programmelemente *paketlokaler* Zugriff erlauben und sprechen abkürzend von *paketlokalen* Programmelementen. Im Java-Jargon ist auch der Terminus *default access* verbreitet, da paketlokaler Zugriff gilt, wenn bei der Deklaration eines Programmelementes keine Zugriffsmodifikatoren angegeben sind.

*öffentlich**paketlokaler  
Zugriff  
default access*

Als ein interessanteres Beispiel zur Diskussion von Kapselung betrachten wir die beiden Pakete des Browsers, insbesondere die Klasse `LinkedList` aus Unterabschn. 2.2.2.1. Gemäß der Paketstruktur von Abb. 2.14 befindet sich die Klasse im Paket `browse.util`. Der Programmtext von Abbildung 2.15 zeigt eine sinnvolle Regelung des Zugriffs auf die beteiligten Programmelemente. Da die Klasse `LinkedList` gerade für die Benutzung in anderen Paketen bestimmt ist, müssen die Klasse selbst, der Konstruktor, die wesentlichen Methoden und die innere Klasse `ListIterator` öffentlich sein. Aus den in Abschn. 2.2.1 genannten Gründen ist es sinnvoll, die Attribute zur Repräsentation der Listenstruktur sowie die dafür benötigte innere Klassendeklaration `Entry` vollständig zu kapseln. Damit sind sie auch vor dem Zugriff von anderen Klassen des Pakets `browse.util` geschützt.

Typische Beispiele für paketlokale Programmelemente sind die Klassen `TextFenster` und `W3Seite`. Sie müssen für die Browser-Klasse zugreifbar sein. Ein Zugriff von außerhalb des Pakets ist aber weder nötig noch erwünscht. Deshalb sollten sie – wie in Abb. 2.14 gezeigt – als paketlokal vereinbart werden, d.h. ohne Zugriffsmodifikator.

Durch die Auszeichnung der öffentlichen Programmelemente und die Kapselung aller anderen Elemente wird für jedes Paket eine Schnittstelle definiert, die aus den öffentlichen Typen und deren öffentlichen Methoden, Konstruktoren und Attributen besteht. Auf alle Elemente dieser Schnittstelle kann aus Klassen, die zu anderen Paketen gehören, zugegriffen werden. Eine Verfeinerung dieser Schnittstellenbildung und der zugehörigen Kapse-

<sup>16</sup>In Kap. 3 werden wir noch den Modifikator `protected` kennen lernen, der zusätzlich zur paketlokalen Benutzung die Benutzung in Unterklassen zulässt.

```
package browse.util;

import browse.ListElem;
import java.util.NoSuchElementException;

public class LinkedList {
    private Entry header = new Entry(null, null, null);
    private int size = 0;

    public LinkedList() { ... }
    public ListElem getLast() { ... }
    public ListElem removeLast() { ... }
    public void addLast(ListElem e) { ... }
    public int size() { ... }

    private static class Entry { ... }

    public class ListIterator {
        private int nextIndex = 0;
        private Entry next = header.next;

        public boolean hasNext() { ... }
        public ListElem next() { ... }
    }

    public ListIterator listIterator() {
        return new ListIterator();
    }
}
```

Abbildung 2.15: Kapselung der Klasse `LinkedList` und der inneren Klassen

lungaspekte werden wir im Zusammenhang mit der Vererbung in Kap. 3 behandeln.

### 2.2.3 Beziehungen zwischen Klassen

In den letzten beiden Abschnitten haben wir gesehen, wie man eine Menge von Klassen strukturieren kann, indem man sie ineinander schachtelt bzw. mehrere Klassen zu Paketen zusammenfasst. In diesem Abschnitt resümieren wir diese Strukturierungsbeziehungen zwischen Klassen und betrachten dann die Beziehungen zwischen Klassen bzw. ihren Objekten von einem allgemeineren Standpunkt.

Eine gute Strukturierung von Klassen richtet sich natürlich nach den Beziehungen, die die Klassen zueinander haben. Typischerweise wird man innere Klassen verwenden, wenn man für die Implementierung einer Klasse Hilfsklassen benötigt, die von außerhalb nicht sichtbar sein sollen. Eine andere typische Anwendung innerer Klassen besteht darin, Objekte bereitzustellen, mit denen man einen komplexeren, aber gekapselten Zugang zu einem Objekt oder einem Objektgeflecht ermöglichen möchte. Als Beispiel dazu haben wir Iteratoren betrachtet, die es gestatten, eine Listenstruktur schrittweise zu durchlaufen, ohne dem Benutzer allgemeinen Zugriff auf die Listenstruktur zu gewähren. Eine Menge von Klassen wird man zu einem Paket zusammenfassen, wenn diese Klassen inhaltlich bzw. anwendungsspezifisch verwandt sind oder einen hinreichend gut abgrenzbaren Teil eines größeren Anwendungsprogramms darstellen.

Wie wir gesehen haben, ist eine gute Strukturierung von Klassen abhängig von den Beziehungen, die zwischen den Klassen und ihren Objekten existieren. Die Festlegung der Klassen und ihrer Beziehungen ist eine der wichtigen Aufgaben des *objektorientierten Entwurfs*. Etwas genauer betrachtet geht es dabei darum,

*objektorientierter Entwurf*

1. die relevanten Objekte des Aufgabenbereichs zu identifizieren, geeignet zu klassifizieren und von allen unnötigen Details zu befreien und
2. ihre Beziehungen so zu entwerfen, dass sich die Dynamik des modellierten Aufgabenbereichs möglichst intuitiv und leicht realisierbar beschreiben lässt.

Der erste der beiden Schritte wird dabei häufig als *Auffinden der Schlüsselabstraktionen* bezeichnet.

Beide Schritte bilden die Grundlage für den Klassenentwurf und haben dementsprechend gewichtigen Einfluss auf die objektorientierte Programmierung. Insbesondere ist es wichtig, sich die Beziehungen zwischen Klassen bzw. ihren Objekten bewusst zu machen, da sie die Strukturierbarkeit, Lesbarkeit und Wartbarkeit von Programmen wesentlich beeinflussen können.

Benutzungs-  
beziehung

Betrachten wir zunächst einmal nur die Beziehungen auf Objektebene. Die einfachste Beziehung zwischen Objekten besteht darin, dass ein Objekt ein anderes Objekt benutzt. Beispielsweise benutzt ein Browser-Objekt einen W3-Server. Man spricht von einer *Benutzungsbeziehung* oder engl. *uses relation*. Wie im Browser-Beispiel kann die Benutzungsbeziehung so angelegt sein, dass ein Objekt von mehreren Objekten benutzt wird. Grundsätzlich kann man sagen, dass ein Objekt  $X$  ein anderes Objekt  $Y$  benutzt, wenn

- eine Methode von  $X$  eine Nachricht an  $Y$  schickt oder
- eine Methode von  $X$  das Objekt  $Y$  erzeugt, als Parameter übergeben bekommt oder als Ergebnis zurückgibt.

hat-ein-  
Beziehung

In vielen Fällen besitzt ein Objekt  $X$ , das ein anderes Objekt  $Y$  benutzt, eine Referenz auf dieses Objekt. Implementierungstechnisch heißt das für Java, dass  $X$  ein Attribut *hat*, das eine Referenz auf  $Y$  enthält. Man sagt deshalb auch oft, dass die Objekte  $X$  und  $Y$  in der *hat-ein-Beziehung* oder englisch *has-a relation* stehen. Eine spezielle Form der hat-ein-Beziehung ist die *Teil-von-Beziehung* oder engl. *part-of relation*. In diesem Fall betrachtet man das referenzierte Objekt als (physikalischen) Teil des referenzierenden Objekts, beispielsweise so, wie ein Motor Teil genau eines Autos ist. Insbesondere kann ein Objekt nicht direkter Teil zweier unterschiedlicher Objekte sein. Da man die Teil-von-Beziehung in Java nur durch Referenzen realisieren kann, muss der Programmierer diese Eigenschaft gewährleisten. In Sprachen wie C++ und BETA ist es möglich, diese Beziehung auch ohne den Umweg über Referenzen zu realisieren.

Teil-von-  
Beziehung

Erreichbar-  
keitsbeziehung

Eine Verallgemeinerung der hat-ein-Beziehung ist die *Erreichbarkeitsbeziehung*: Ein Objekt  $X$  *erreicht* ein Objekt  $Y$ , wenn man durch wiederholtes Selektieren eines Attributs von  $X$  nach  $Y$  kommen kann. Die Erreichbarkeitsbeziehung ist wichtig, um abzuschätzen, welche Fernwirkungen die Modifikation eines Objekts haben kann: Wird ein Objekt  $Y$  verändert, kann sich das Verhalten aller Objekte ändern, die  $Y$  erreichen (beispielsweise, weil sie bei einem Methodenaufruf den Zustand von  $Y$  lesen). Andersherum gilt, dass ein Methodenaufruf maximal diejenigen Objekte verändern kann, die vom impliziten und von den expliziten Parametern sowie den zugreifbaren Klassenattributen aus erreichbar sind.

Viele Aspekte der Beziehungen zwischen Objekten lassen sich auf ihre Klassen übertragen. So sagen wir, dass eine Klasse  $A$  eine andere Klasse  $B$  *benutzt*, wenn eine Methode von  $A$  eine Nachricht an  $B$ -Objekte schickt oder wenn eine Methode von  $A$   $B$ -Objekte erzeugt, als Parameter übergeben bekommt oder als Ergebnis zurückgibt. Entsprechend kann man die hat-ein- und die Erreichbarkeitsbeziehung auf Klassen definieren. Ein guter Klassendesign zeichnet sich dadurch aus, dass die Anzahl der Benutzungsbeziehungen gering und überschaubar gehalten wird und weitgehend Gebrauch

von Kapselungstechniken gemacht wird, um die Benutzungsschnittstellen sauber herauszuarbeiten.

Neben der Benutzungs- und hat-ein-Beziehung spielt in der objektorientierten Programmierung die *ist-ein-Beziehung* zwischen Objekten und Typen und die Subtypbeziehung auf den Typen eine zentrale Rolle. Wenn B ein Subtyp von A ist, so ist jedes Objekt vom Typ B auch ein Objekt vom Typ A. Diese Art von Beziehungen werden wir im folgenden Kap. 3 kennen lernen.

*ist-ein-  
Beziehung*





## Selbsttestaufgaben

### Aufgabe 1: Eine Klasse Wochentag

Betrachten Sie das folgende Programmfragment eines Typen für Wochentage:

```
public class Wochentag {
    private int tag;

    public Wochentag() { ... }
    public void setTag(int i) throws KeinTagException { ... }
    public int getTag() { ... }
    public void naechsterTag() { ... }
    public void vorhergehenderTag() { ... }
    public String toString() { ... }
}
```

Vervollständigen Sie die Implementierung so, dass die folgenden Eigenschaften gelten! Vereinbaren Sie bei der Implementierung in der Klasse Wochentag keine neuen Attribute!

1. Der Konstruktor soll ein Wochentag-Objekt erzeugen und den aktuellen Wochentag auf Montag setzen.
2. Die Methode `naechsterTag()` soll ein Wochentag-Objekt auf den folgenden Wochentag setzen. (Die Wochentage werden zyklisch behandelt!)
3. Die Methode `vorhergehenderTag()` soll ein Wochentag-Objekt auf den vorhergehenden Wochentag setzen.
4. Die Methode `setTag(int i)` soll den Wochentag des Wochentag-Objektes setzen (0 für Montag, 1 für Dienstag, usw.). Falls ein `int`-Wert kleiner Null oder größer sechs übergeben wird, soll eine `KeinTagException` geworfen werden. Vereinbaren Sie eine Klasse `KeinTagException`. `KeinTagException` soll als der Klasse `Exception` untergeordnete Klasse deklariert sein.
5. Die Methode `getTag()` soll den Tag codiert als `int`-Wert als Ergebnis liefern.
6. Die `toString()`-Methode soll den Namen des Wochentages als `String` zurückliefern.

## Aufgabe 2: Objektgeflecht und Überladung

Betrachten Sie folgenden Programmcode:

```
public class Objektgeflecht {
    Objektgeflecht a, b, c;

    public Objektgeflecht () {
        a = null;
        b = null;
        c = null;
    }

    public Objektgeflecht (Objektgeflecht a,
                           Objektgeflecht b,
                           Objektgeflecht c) {

        this.a = a;
        this.b = b;
        this.c = c;
    }

    public static void main (String argv[]) {
        Objektgeflecht u = new Objektgeflecht();
        Objektgeflecht v = new Objektgeflecht();
        Objektgeflecht w = new Objektgeflecht(u, v, null);

        (w.a).b = v;
        v.a = u.b;
        (u.b).c = w;
        w.c = v.c;
        u.c = (v.a).c;
        /* Markierung */
    }
}
```

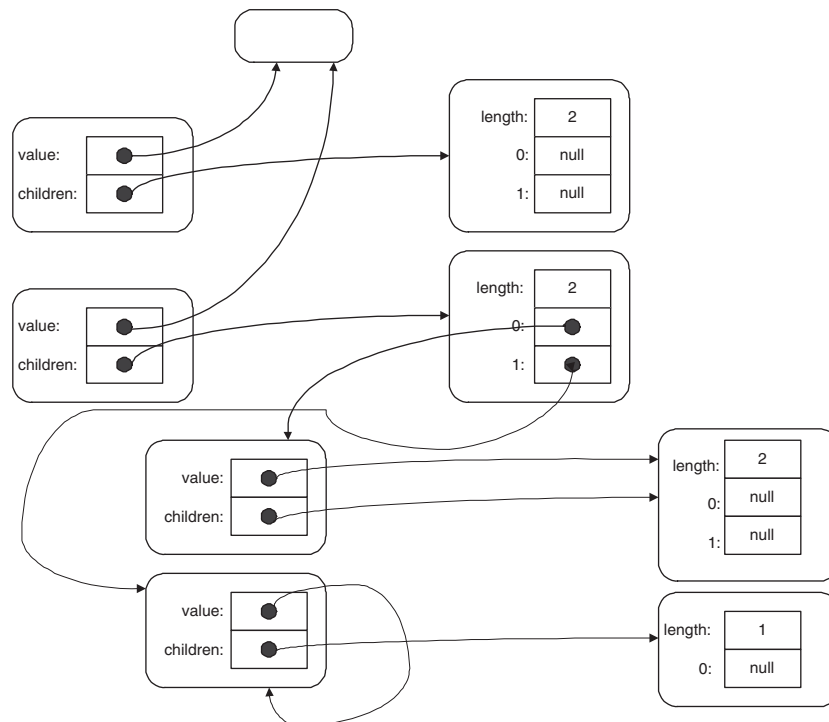
1. Bei welchen Deklarationselementen tritt Überladung auf und bei welchen Anweisungen muss der Compiler eine Überladung auflösen? Warum gelingt die Auflösung?
2. Zeichnen Sie das Objektgeflecht, das entstanden ist, wenn die Ausführung der `main`-Methode die markierte Stelle erreicht hat.

## Aufgabe 3: Eine Klasse Tree

Gegeben sei die Klasse `Tree`, mit deren Hilfe baumartige Datenstrukturen aufgebaut werden können. Knoten dieser Strukturen, also Objekte der Klasse `Tree`, enthalten als Nutzinformation ein `value`-Attribut, mit dessen Hilfe ein `Tree`-Knoten auf ein beliebiges anderes Objekt verweisen kann.

```
class Tree {
    Object value;
    Tree[] children;
}
```

Ergänzen Sie die gegebene Klasse `Tree` um eine Methode mit der Signatur `public static void main(String[] args)` (und ggf. weitere Methoden) so, dass nach Ausführung der `main`-Methode das folgende Objektgeflecht entstanden ist! (Das leer dargestellte Objekt stellt dabei ein Objekt der Klasse `Object` dar.)



#### Aufgabe 4: Pakete

Ergänzen Sie die folgenden Klassenimplementierungen so, dass die folgenden Eigenschaften erfüllt sind:

1. Jede Klasse soll sich in einem eigenen, benannten Paket befinden. Dabei sollen die Pfade, die den Paketen von `ObjectStore` und `NumberStrings` entsprechen, unterhalb des Pfades liegen, der dem Paket entspricht, das die Klasse `Top` enthält.
2. Die `size`-Methode der Klasse `ObjectStore` soll nur in der Klasse `ObjectStore` zugreifbar sein.
3. Die Datei, in der die Klasse `Top` steht, soll keine `import`-Anweisungen für das Paket enthalten, zu dem die Klasse `NumberStrings` gehört.

4. Die drei Klassen zusammen sollen ein korrektes Java-Programm darstellen.

```
import java.util.*;

class Top {
    public static void main(String[] argv) {
        ObjectStore store = new ObjectStore();
        for (int i = 0; i < argv.length; i++) {
            store.add(argv[i]);
            NumberStrings.size++;
        }
        System.out.println("Anzahl gespeicherte Strings : "
                           + NumberStrings.size);
        for (int i = 0; i < 10; i++) store.add(new Integer(i));
    }
}

class ObjectStore {
    Vector<Object> v;
    ObjectStore() { v = new Vector<Object> (); }

    void add(Object o) {
        v.add(o);
        System.out.println("Anzahl gespeicherter Objekte : "
                           + size());
    }

    int size() { return v.size(); }
}

class NumberStrings {
    static int size = 0;
}
```

### Aufgabe 5: Parametrische Klassen mit inneren Klassen

Wie muss die Implementierung der auf Seite 121 gezeigten Klasse `LinkedList<ET>` abgeändert werden, damit ihre innere Klasse `Entry` keinen Typparameter mehr benötigt, aber trotzdem sichergestellt bleibt, dass ein `Entry`-Objekt nur Elemente vom Typ `ET` enthält? Ändern Sie den Quellcode entsprechend ab und markieren Sie alle Zeilen mit `'/**/'`, die von dieser Änderung betroffen sind.

## Musterlösungen zu den Selbsttestaufgaben

### Aufgabe 1: Eine Klasse Wochentag

Die folgende Implementierung der Klasse Wochentag genügt den Anforderungen der Aufgabenstellung.

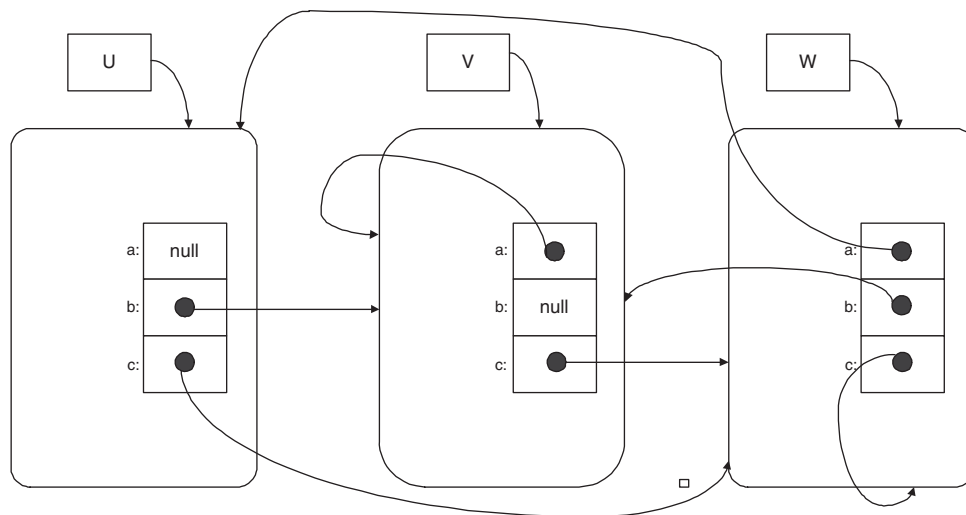
```
public class Wochentag {
    private int tag;

    public Wochentag() {
        tag = 0;
    }
    public void setTag(int i) throws KeinTagException {
        if (i < 0 || i > 6)
            throw new KeinTagException();
        else
            tag = i;
    }
    public int getTag() {
        return tag;
    }
    public void naechsterTag() {
        tag = (tag + 1) % 7;
    }
    public void vorhergehenderTag() {
        if (tag == 0)
            tag = 6;
        else
            tag--;
    }
    public String toString() {
        switch(tag) {
            case 0: return "Montag";
            case 1: return "Dienstag";
            case 2: return "Mittwoch";
            case 3: return "Donnerstag";
            case 4: return "Freitag";
            case 5: return "Samstag";
            case 6: return "Sonntag";
            default: return "ERROR";
        }
    }
}

class KeinTagException extends Exception {
}
```

### Aufgabe 2: Objektgeflecht und Überladung

1. Der Konstruktornamen der Klasse `Objektgeflecht` ist überladen. Der erste Konstruktor ist parameterlos, der zweite Konstruktor besitzt drei Parameter vom Typ `Objektgeflecht`. Auflösung ist nötig in den drei ersten Anweisungen der `main`-Methode, in denen der Konstruktor aufgerufen wird. Der Compiler kann die Aufrufe korrekt auflösen, da sich beide Konstrukturen in der Anzahl ihrer Parameter unterscheiden.
2. Wenn die Ausführung der `main`-Methode die durch `/* Markierung */` gekennzeichnete Stelle erreicht hat, ist das folgende Objektgeflecht entstanden.



### Aufgabe 3: Eine Klasse Tree

Die `main`-Methode des folgenden Java-Programms erzeugt das in der Aufgabe angegebene Objektgeflecht im Objektspeicher. Beachten Sie, dass nicht alle Komponenten der `children`-Attribute mit Werten initialisiert werden. Prüfen Sie durch einfache Ausgabeanweisungen, mit welchen Werten diese Feldkomponenten automatisch besetzt werden.

```
class Tree {
    Object value;
    Tree[] children;

    public static void main(String[] argv) {
        Tree tree1 = new Tree();
        Tree tree2 = new Tree();
        Tree tree3 = new Tree();
        Tree tree4 = new Tree();
        tree1.children = new Tree[2];
```

```

        tree2.children = new Tree[2];
        tree3.children = new Tree[2];
        tree4.children = new Tree[1];
        Object o = new Object();
        tree1.value = o;
        tree2.value = o;
        tree3.value = tree3.children;
        tree4.value = tree4;
        tree1.children[0];
        tree1.children[1];
        tree2.children[0] = tree3;
        tree2.children[1] = tree4;
    }
}

```

#### Aufgabe 4: Pakete

Das folgende Programm erfüllt die gestellten Anforderungen:

```

package p_top;

import p_top.p_objectStore.*;
class Top {
    public static void main (String[] argv) {
        ObjectStore store = new ObjectStore();
        for (int i = 0; i < argv.length; i++) {
            store.add (argv[i]);
            p_top.p_numberStrings.NumberStrings.size++;
        }
        System.out.println("Anzahl gespeicherte Strings : " +
                           p_top.p_numberStrings.NumberStrings.size);
        for ( int i = 0; i < 10; i++) store.add (new Integer(i));
    }
}

package p_top.p_objectStore;

import java.util.*;
public class ObjectStore {

    Vector<Object> v;
    public ObjectStore() {
        v = new Vector<Object>();
    }

    public void add (Object o) {
        v.add(o);
    }
}

```



```

        System.out.println("Anzahl gespeicherter Objekte :  "
                           + size());
    }

    private int size () {
        return v.size();
    }
}

package p_top.p_numberStrings;

public class NumberStrings {
    public static int size = 0;
}

```

1. Damit jede Klasse in einem eigenen Paket steht, muss in der JDK-Umgebung für jedes Paket ein eigenes Unterverzeichnis im Dateisystem angelegt werden und die CLASSPATH-Variable muss auf das (die) Verzeichnis(se) zeigen, in dem die Paketverzeichnisse liegen. In unserem Fall werden die Unterverzeichnisse `p_top`, `p_top/p_objectStore` und `p_top/p_numberStrings` erzeugt und die Dateien `Top.java`, `ObjectStore.java` und `NumberStrings.java` darin untergebracht. Jede Datei erhält zudem die notwendige package-Anweisung.
2. Damit die Methode `size` nur in der Klasse `ObjectStore` zugreifbar ist, muss sie mit dem Modifikator `private` versehen werden.
3. Da die Klasse `Top` die Klasse `NumberStrings` benutzt, die Datei `Top.java` jedoch keine `import`-Anweisungen für diese Klasse benutzen soll, muss beim Benutzen der Klasse `NumberStrings` der voll qualifizierte Name angegeben werden.
4. Die Klassen, die in anderen Paketen benutzt werden sollen, müssen mit dem Modifikator `public` gekennzeichnet werden.
5. Die Methoden, Konstruktoren und Attribute, die aus anderen Paketen benutzt werden sollen, müssen mit dem Modifikator `public` gekennzeichnet werden.<sup>17</sup>
6. In der Datei `ObjectStore.java` muss eine `import`-Anweisung für die Klasse `Vector` aus dem Paket `java.util` eingefügt werden.

---

<sup>17</sup> Der Modifikator `protected` wird später eingeführt.

**Aufgabe 5: Parametrische Klassen mit inneren Klassen**

Die innere Klasse `Entry` kann ohne eigenen Typparameter auskommen, wenn sie als **nicht** statische innere Klasse deklariert wird. Dann kann auch an allen Stellen, wo `Entry` verwendet wird, der Typparameter entfallen. Folgender Quellcode zeigt die entsprechend abgeänderte Version von `LinkedList<ET>`. Die geänderten Zeilen sind durch vorangestelltes `/**/` gekennzeichnet.

```
class LinkedList<ET> {

    /**/ class Entry {
        ET element;
    /**/     Entry next;
    /**/     Entry previous;

    /**/     Entry(ET element, Entry next, Entry previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
        }
    }

    /**/ Entry header = new Entry(null, null, null);
    int size = 0;

    // Constructs an empty Linked List.
    LinkedList() {
        header.next = header;
        header.previous = header;
    }

    // Returns the last Element in this List.
    ET getLast() {
        if( size==0 ) throw new NoSuchElementException();
        return header.previous.element;
    }

    // Removes and returns the last Element from this List.
    ET removeLast() {
    /**/     Entry lastentry = header.previous;
        if(lastentry==header) throw new NoSuchElementException();
        lastentry.previous.next = lastentry.next;
        lastentry.next.previous = lastentry.previous;
        size--;
        return lastentry.element;
    }

    // Appends the given element to the end of this List.
    void addLast(ET e) {
    /**/     Entry newEntry = new Entry(e, header, header.previous);
        header.previous.next = newEntry;
    }
```

```
        header.previous = newEntry;
        size++;
    }

    // Returns the number of elements in this List.
    int size() {
        return size;
    }

    class ListIterator{
        private int nextIndex = 0;
    /**/ private Entry next = header.next;

        boolean hasNext() {
            return nextIndex != size;
        }
        ET next() {
            if (nextIndex == size)
                throw new NoSuchElementException();
            ET elem = next.element;
            next = next.next;
            nextIndex++;
            return elem;
        }
    }

    ListIterator listIterator() {
        return new ListIterator();
    }
}
```

# Studierhinweise zur Kurseinheit 3

Diese Kurseinheit beschäftigt sich mit den ersten beiden Abschnitten vom dritten Kapitel des Kurstexts. Im Mittelpunkt dieser Kurseinheit steht die Behandlung von Subtyping, einem zentralen Konzept der objektorientierten Programmierung. Sie sollten den gesamten Text zu dieser Kurseinheit im Detail studieren und verstehen. Versuchen Sie, selbst Schnittstellentypen zu entwerfen und mit diesen kleine Programme zu realisieren. Prüfen Sie nach dem Durcharbeiten des Kurstextes, ob Sie die Lernziele erreicht haben. Arbeiten Sie dazu auch die Selbsttestaufgaben am Ende der Kurseinheit durch.

## Lernziele:

- Zentrale Begriffe wie Klassifikation, Abstraktion, Spezialisierung, „ist-ein“-Beziehung.
- Subtyping: das allgemeine Konzept.
- Subtyping und Schnittstellentypen in Java.
- Dynamische Methodenauswahl: Was ist das? Wofür ist das gut?
- Syntaktische Bedingungen zwischen Subtypen und konformes Verhalten von Subtypen.
- Parametrische Typen und Subtyping.
- Polymorphie: Was ist das? Welche Arten werden unterschieden?
- Programmieren mit Schnittstellen.
- Programmieren mit Aufzählungstypen.



# Kapitel 3

## Vererbung und Subtyping

Dieses Kapitel erläutert den Zusammenhang zwischen objektorientierter Programmierung und Klassifikationen und behandelt Subtyping und Vererbung. Damit vervollständigt es die Beschreibung der Grundkonzepte objektorientierter Programmierung (vgl. Abschn. 1.4).

Klassifikationen dienen der Strukturierung von Dingen und Begriffen in vielen Bereichen des täglichen Lebens und der Wissenschaft. Der erste Abschnitt dieses Kapitels zeigt die Analogie zwischen derartigen allgemeinen Klassifikationen und den hierarchischen Klassenstrukturen in der objektorientierten Programmierung auf. Davon ausgehend wird erläutert, was Abstrahieren und Spezialisieren von Objekten bzw. Klassen bedeutet. Der zweite Abschnitt des Kapitels behandelt Subtyping und das Schnittstellenkonstrukt von Java. Der dritte Abschnitt beschreibt Vererbung. Schließlich wird kurz auf den Einsatz objektorientierter Techniken zur Wiederverwendung von Programmteilen eingegangen.

### 3.1 Klassifizieren von Objekten

Objekte besitzen eine klar definierte Schnittstelle. Die Schnittstelle gibt darüber Auskunft, welche Methoden und Attribute ein Objekt zur Verfügung stellt. Schnittstellen bilden eine hervorragende Grundlage, um Objekte gemäß ihren Fähigkeiten klassifizieren zu können. Dieser Abschnitt erläutert, was Klassifikationen sind, wie man sie erweitert, inwiefern sie in der Programmierung eine Rolle spielen, was Abstraktion in der Programmierung bedeuten kann und inwiefern Spezialisieren ein zentraler Aspekt bei der Wiederverwendung ist.

**Klassifikation.** Klassifizieren ist eine allgemeine Technik, um Wissen über Begriffe, Dinge und deren Eigenschaften zu strukturieren. Typisch dabei sind hierarchische Strukturen. Erste Beispiele von Klassifikationen haben wir be-

reits in den Abschnitten 1.1.1 und 1.2.3 kennen gelernt (vgl. die Abbildungen 1.1 und 1.4). Um die weite Verbreitung von Klassifikationen und ihre grundlegende Bedeutung zu illustrieren, gibt Abbildung 3.1 Beispiele aus den Rechtswissenschaften, der Zoologie, Instrumentenkunde, Geometrie und eine Klassifikation für Wiedergabegeräte. An den Beispielen können wir sehen,

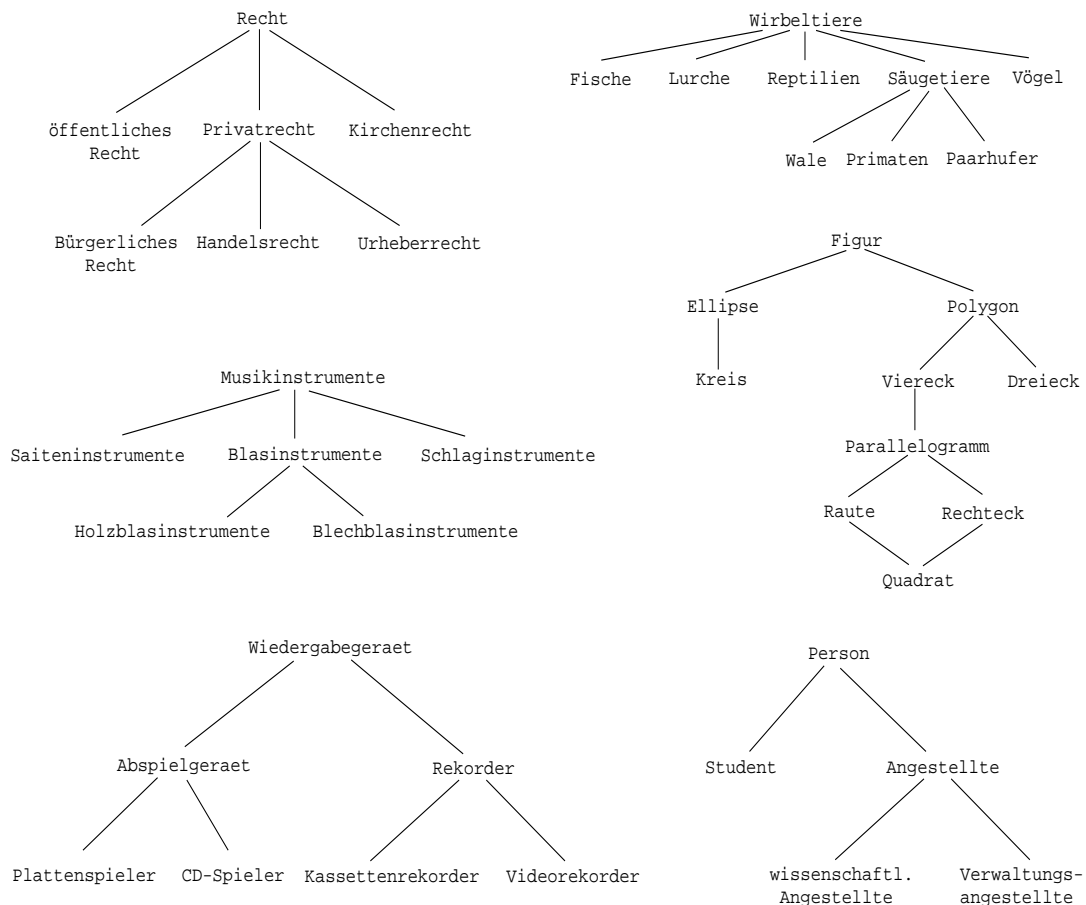


Abbildung 3.1: Beispiele für Klassifikationen

dass Klassifikationen recht grundlegenden Charakter haben können, vielfach aber auch nur zur Strukturierung von Begriffen des täglichen Lebens dienen.

In der objektorientierten Programmierung wird der Klassifikationsmechanismus zum Strukturieren von Anwendungsprogrammen und von Programmbibliotheken eingesetzt. Beispielsweise wird er in der Standardbibliothek von Java genutzt, um die unterschiedlichen Ausnahmen und die verschiedenen Ströme für Ein- und Ausgabe zu strukturieren (vgl. Kap. 4) sowie um die verschiedenen Oberflächenkomponenten und möglichen Ereignisse zu klassifizieren (vgl. Kap. 5).

In Klassifikationen stehen üblicherweise die allgemeineren Begriffe oben, die spezielleren unten, die Begriffe bilden also eine Hierarchie. In den

meisten Fällen werden dabei Objekte aus einem bestimmten Wissens-/Anwendungsbereich klassifiziert (z.B. Gesetze, Tiere, Instrumente, Geräte). Die allgemeineren Klassen umfassen dabei alle Objekte der spezielleren Klassen. Andersherum ausgedrückt: Jedes Objekt einer spezielleren Klasse *ist* auch *ein* Objekt aller seiner allgemeineren Klassen: Ein Holzblasinstrument *ist ein* Blasinstrument und natürlich auch ein Musikinstrument; ein Quadrat ist eine Raute. „A ist ein B“ sagt im Wesentlichen aus, dass A alle Eigenschaften von B hat – und ggf. einige zusätzliche. In diesem Sinne ist A spezieller als B. Eine derartige *ist-ein-Beziehung* (englisch: *is-a relation*) ist ein wichtiges Merkmal einer guten Klassifikation. Klassifizierende Begriffe werden in der objektorientierten Programmierung i.A. auf Klassen abgebildet, die selbst entsprechend der Hierarchie auf den zugeordneten Begriffen angeordnet sind. Wie wir sehen werden, führt nicht jede Klassenhierarchie in der objektorientierten Programmierung automatisch zu einer *ist-ein-Beziehung* auf den Objekten. Es lassen sich nämlich durchaus Programme realisieren, bei denen sich Objekte von Klassen weiter unten in der Hierarchie völlig anders verhalten als Objekte von Klassen weiter oben in der Hierarchie.

*ist-ein-  
Beziehung*

Die Möglichkeit, Objekte klassifizieren zu können und die Implementierung von Objekten entsprechend der Klassifikation strukturieren zu können, ist eine zentrale Stärke der objektorientierten Programmiersprachen. Objekte bzw. ihre Klassen werden gemäß ihrer Schnittstelleneigenschaften klassifiziert. Eine Klasse  $K_1$  ist spezieller als eine andere Klasse  $K_2$ , wenn sie bzw. ihre Objekte mindestens die Methoden und Attribute von  $K_2$  besitzen.  $K_1$  und  $K_2$  sind hierarchisch zueinander angeordnet.  $K_1$  steht als speziellere Klasse in der Klassenhierarchie weiter unten als die allgemeinere Klasse  $K_2$ . Speziellere Objekte können anstelle allgemeinerer Objekte verwendet werden. Klassifikationen von Objekten werden in der objektorientierten Programmierung also in Form von Klassen- bzw. Typhierarchien ausgedrückt. Insbesondere stellen die meisten objektorientierten Sprachen eine allgemeinste Klasse zur Verfügung; in Java ist das die Klasse `Object`.

Ein zentraler Aspekt der objektorientierten Programmentwicklung ist der Entwurf und die Realisierung von geeigneten Klassen- bzw. Typhierarchien. Dabei stehen zwei Aufgaben im Vordergrund:

1. das Abstrahieren, d.h. Verallgemeinern von Typen;
2. das Spezialisieren von Klassen bzw. Typen.

Spezialisierung wird häufig angewendet, um existierende Klassen- bzw. Typhierarchien zu erweitern, z.B. die Standardbibliotheken der benutzten Programmiersprache oder anwendungsspezifische Klassenbibliotheken.

Bevor wir uns in den nächsten Abschnitten mit der programmiersprachlichen Unterstützung dieser Techniken befassen, wollen wir sie hier an zwei kleinen Beispielen illustrieren.



*Abstraktion*

**Abstraktion.** Meyers großes Taschenlexikon definiert *Abstraktion* als „das Heraussondern des unter einem bestimmten Gesichtspunkt Wesentlichen vom Unwesentlichen, Zufälligen sowie das Ergebnis dieses Heraussonderns“. Das klingt zunächst einmal recht einfach. In der Anwendung ist es aber häufig schwierig, das Wesentliche und damit die geeigneten Abstraktionen zu finden. Beispielsweise hätte eine Klassifikation der Tiere in behaarte und nicht-behaarte oder gemäß der Anzahl ihrer Beine wenig Erfolg gehabt. Auch in der softwaretechnischen Praxis ist es oft gar nicht trivial, die wesentlichen Gemeinsamkeiten zu erkennen. Beispielsweise war es ein großer Fortschritt von Unix, eine gemeinsame Schnittstelle für zum Teil sehr unterschiedliche Ein- und Ausgabegeräte bereitzustellen.

Um die programmtechnischen Aspekte fürs Abstrahieren studieren zu können, betrachten wir ein Beispiel, bei dem klar ist, wie die Abstraktion aussehen soll. Wir nehmen an, dass wir etliche Klassen besitzen, in denen eine Methode `drucken` existiert (vgl. dazu die in Abschn. 1.2.3, S. 22, diskutierte Pascal-Fassung):

```
class Student ... { ... public void drucken(){...} ... }
class WissAng ... { ... public void drucken(){...} ... }
class VerwAng ... { ... public void drucken(){...} ... }
...
class Skript ... { ... public void drucken(){...} ... }
```

Um die gemeinsame Eigenschaft dieser Klassen auszudrücken (nämlich, dass sie alle die Methode `drucken` besitzen), sind in Java zwei Dinge nötig. Erstens muss ein Typ vereinbart werden, zu dem alle Objekte mit einer Methode `drucken` gehören sollen; wir geben dem Typ den Namen `Druckbar`. Die entsprechende Typdeklaration hat in Java folgende Form<sup>1</sup>:

```
interface Druckbar {
    void drucken();
}
```

Zweitens muss man in Java explizit<sup>2</sup> angeben, welche Objekte zu diesem Typ gehören sollen. Diese Angabe erfolgt zusammen mit der Klassendeklaration durch Verwendung des Schlüsselworts `implements` gefolgt von dem Typnamen. Wir demonstrieren das am Beispiel `Student`:

<sup>1</sup>Man könnte statt einer Schnittstellendeklaration auch eine abstrakte Klasse deklarieren (s. Seite 166). Das würde aber, wie wir später noch sehen werden, die Möglichkeiten zum Erben in Java zu sehr einschränken.

<sup>2</sup>Alternativ könnte man implizit jedes Objekt *X* zum Typ `Druckbar` rechnen, das eine Methode `drucken` besitzt; diese Information ließe sich automatisch aus der Klassendeklaration zu *X* ableiten.

```

class Student implements Druckbar {
    ...
    public void drucken(){...}
}

```

Um zu zeigen, wie der abstrakte Typ `Druckbar` eingesetzt werden kann, implementieren wir einen Behälter für druckbare Objekte. Dafür benutzen wir unsere parametrische Listenklasse mit Listeneratoren (vgl. Abschn. 2.1.6, S. 104, und Abschn. 2.2.2.1, S. 121). Der Behälter soll eine Methode `alle_drucken` besitzen, mit der alle seine Elemente gedruckt werden können (vgl. auch Abschnitt 1.2.3, S. 22):

```

class Behaelter {
    LinkedList<Druckbar> dieElemente;
    ...
    public void alle_drucken() {
        // Initialisieren des Iterators
        LinkedList<Druckbar>.ListIterator it =
                                dieElemente.listIterator();
        // drucken aller Elemente im Behaelter
        while( it.hasNext() ) {
            Druckbar e = it.next();
            e.drucken();
        }
    }
}

```

Der Variablen `e` werden der Reihe nach alle Elemente des Behälters zugewiesen. Dabei kann sie ganz unterschiedliche Objekte referenzieren, insbesondere z.B. `Student`-Objekte und `Skript`-Objekte. Im Sinne des objektorientierten Grundmodells aus Kapitel 1 würde man dann sagen, dass diesen Objekten mittels der Anweisung `e.drucken()` die Nachricht `drucken` geschickt wird und die Objekte darauf mit der Ausführung ihrer eigenen Druckmethode reagieren. Da diese Sprechweise aber etwas länglich ist, werden wir meist nur davon sprechen, dass „die“ Methode `drucken` aufgerufen wird. Es sollte aber klar sein, dass dies eine saloppe Sprechweise ist, da je nach dem von `e` referenzierten Objekt unterschiedliche Methoden aufgerufen werden. Man beachte im Übrigen, dass die Benutzung einer Liste vom Typ `LinkedList<Druckbar>` garantiert, dass der Variablen `e` nur Elemente zugewiesen werden, die vom Typ `Druckbar` oder Subtypen davon sind (s. auch Abschnitt 2.1.6.1). In `LinkedList<Druckbar>` gibt `<Druckbar>` ja den Typ der Listenelemente an. Andere Typen, die nicht Subtypen von `Druckbar` sind, können daher nicht in der Liste gespeichert werden.

Wenn wir die Java-Fassung mit der Pascal-Formulierung von Kap. 1, S. 22, vergleichen, ergeben sich drei wichtige Unterschiede:

1. Die Fallunterscheidung, welche Druckmethode auszuführen ist, muss in Java nicht explizit formuliert werden wie in Pascal, sondern geschieht automatisch je nach dem Typ des Objekts, das von `e` referenziert wird. Da dieses Binden des Methodennamens an die Methodenimplementierungen nur zur Laufzeit des Programms erfolgen kann (warum?), spricht man von dynamischem Binden oder dynamischer Methodenwahl (vgl. Abschn. 3.2).
2. In Pascal legt die Typdeklaration von `Druckbar` fest, welche Objekte druckbar sind. In Java deklariert jeder Typ für sich, ob er den Typ `Druckbar` implementiert (durch Angabe von `implements Druckbar`). Wenn er es tut, muss er natürlich eine Methode `drucken` besitzen.
3. In Java können Objekte zu mehreren Typen gehören; z.B. ist ein `Student`-Objekt vom Typ `Student` und vom Typ `Druckbar` und kann insbesondere direkt Variablen vom Typ `Druckbar` zugewiesen werden. In Pascal ist dazu immer ein expliziter Selektionsschritt nötig (im Pascal-Fragment von Kap. 1 war dies `e.s`).

Die Technik, mit der die objektorientierte Programmierung die skizzierte Aufgabenstellung behandelt, bringt im Zusammenhang mit Wiederverwendung deutliche Vorteile mit sich: Beim Hinzufügen neuer Klassen für druckbare Objekte braucht weder die Typdeklaration von `Druckbar` noch die Implementierung der Methode `alle_drucken` in der Klasse `Behaelter` angepasst zu werden; beide funktionieren auch für die Objekte der hinzugefügten Klassen.

#### Spezialisierung

**Spezialisierung.** Unter *Spezialisierung* verstehen wir das Hinzufügen „speziellerer“ Eigenschaften zu einem gegebenen Gegenstand oder das Verfeinern eines Begriffs durch das Einführen von Unterscheidungen. In ähnlicher Weise spricht man etwa von beruflicher Spezialisierung und meint damit, dass eine allgemeine berufliche Qualifikation im Hinblick auf spezielle *zusätzliche* Fähigkeiten erweitert wird<sup>3</sup>. In der objektorientierten Programmierung ist Spezialisierung die zentrale Technik, um existierende Programme auf die eigenen Bedürfnisse und auf neue Anforderungen anzupassen. Typische Beispiele liefern Baukästen (englisch: tool kits) bzw. Programmgerüste (englisch: frameworks) zur Implementierung graphischer Bedienoberflächen; sie stellen Grundkomponenten (Fenster, Schaltflächen, Texteditoren, Layout-Manager) zur Verfügung, die der Oberflächenprogrammierer durch Spezialisierung an

<sup>3</sup>Sowohl im allgemeinen Sprachgebrauch als auch im Zusammenhang mit objektorientierter Programmierung wird Spezialisierung teilweise auch mit einer etwas anderen Bedeutung verwendet, nämlich in dem Sinne, dass darunter das *Einschränken* allgemeiner Fähigkeiten im Hinblick auf eine speziellere Aufgabe verstanden wird (vgl. z.B. [Goo99]).

seine Anforderungen anpassen und erweitern kann. Dabei brauchen nur die neuen, speziellen Eigenschaften programmiert zu werden. Die Implementierungen für die Standardeigenschaften werden von den Grundkomponenten geerbt.

Derartiges Vererben von Programmteilen zur Spezialisierung von Implementierungen zu unterstützen ist eine herausragende Fähigkeit objektorientierter Programmiersprachen. Vererbung geht dabei in zwei Aspekten über systematisches Kopieren und Einfügen von Programmtexten hinaus:

1. Es findet kein Kopieren statt, d.h. ein Programmteil, der an mehrere Spezialisierungen vererbt wird, ist nur einmal vorhanden, was sich positiv auf die Größe der ausführbaren Programme auswirkt und die Wartbarkeit der Programme verbessert.
2. Es können auch Programmteile geerbt werden, die nur in anderen Programmiersprachen oder nur in Maschinensprache vorliegen.

Im Übrigen kann ein Anwender Programmteile erben, ohne dass der Softwarehersteller diese Programmteile offen legen muss.

Um die wesentlichen programmtechnischen Aspekte im Zusammenhang von Spezialisierung und Vererbung hier diskutieren zu können, betrachten wir wieder ein kleines Beispiel. Das Beispiel ist etwas künstlich, zeigt aber alle wichtigen Aspekte von Vererbung. In Java sind Oberflächenfenster, die in keinem anderen Fenster enthalten sind, Objekte der Klasse `Frame` (siehe Kap. 5). Jeder `Frame` hat eine Hintergrundfarbe, die sich mit den Methoden `getBackground` und `setBackground` abfragen und einstellen lässt. Eine neu eingestellte Hintergrundfarbe wird am Bildschirm angezeigt, sobald der `Frame` neu gezeichnet wird. `Frames` sollen nun so spezialisiert werden, dass sie sich ihre letzte Hintergrundfarbe merken und eine Methode anbieten, um die letzte Hintergrundfarbe zur aktuellen Hintergrundfarbe zu machen. Dies erreicht man durch die drei typischen Spezialisierungsoperationen:

1. Hinzufügen von Attributen: Wir fügen ein Attribut `letzterHintergrund` vom Typ `Color` hinzu, in dem wir die letzte Hintergrundfarbe speichern können.
2. Hinzufügen von Methoden: Wir erweitern die Klasse um eine Methode `einstellenLetztenHintergrund`.
3. Modifizieren von Methoden der zu spezialisierenden Klasse: Wir müssen die Methode `setBackground` der Klasse `Frame` so modifizieren, dass sie die letzte Hintergrundfarbe im neuen Attribut speichert.

Die spezialisierte Klasse nennen wir `MemoFrame` (siehe Abb. 3.2).

```
import java.awt.* ;

class MemoFrame extends Frame {
    private Color letzterHintergrund;

    public void einstellenLetztenHintergrund() {
        setBackground( letzterHintergrund );
    }
    public void setBackground( Color c ) {
        letzterHintergrund = getBackground();
        super.setBackground( c );
    }
}

public class TestMemoFrame {
    public static void main(String[] args) {
        MemoFrame f = new MemoFrame();

        // Fenstergroesse auf 300x200-Pixel setzen
        f.setSize( 300, 200 );
        f.setVisible( true );

        // Hintergrundfarbe auf rot setzen
        f.setBackground( Color.red );
        f.update( f.getGraphics() );
        // 4 Sekunden warten
        try{ Thread.sleep(4000); } catch( Exception e ){}

        // Hintergrundfarbe auf gruen setzen
        f.setBackground( Color.green );
        f.update( f.getGraphics() );
        try{ Thread.sleep(4000); } catch( Exception e ){}

        // Letzte Hintergrundfarbe wieder zur Aktuellen machen
        f.einstellenLetztenHintergrund();
        f.update( f.getGraphics() );
        try{ Thread.sleep(4000); } catch( Exception e ){}

        // Fenster schliessen
        System.exit( 0 );
    }
}
```

Abbildung 3.2: Klasse MemoFrame mit Testrahmen

Sie erweitert die Klasse `Frame`, d.h. sie erbt von ihr alle Methoden und Attribute. Dies deklariert man in Java durch das Schlüsselwort `extends` gefolgt vom Typnamen. Zusätzlich enthält sie das Attribut `letzterHintergrund` und die Methode `einstellenLetztenHintergrund`. Die erweiterte Funktionalität der Methode `setBackground` erhält man dadurch, dass man eine Methode gleichen Namens deklariert, die den aktuellen Hintergrund im Attribut `letzterHintergrund` speichert und dann die Methode `setBackground` von Klasse `Frame` aufruft, um den Hintergrund zu ändern (`super.setBackground(c)`). Wir verändern hier also das Verhalten einer Methode, deren Implementierung wir nicht kennen. Insbesondere benötigen wir kein Wissen darüber, wie und wann der Hintergrund eines Fensters zu modifizieren ist.

(Abbildung 3.2 zeigt im Übrigen auch eine Klasse zum Testen von `MemoFrame`; sie erzeugt ein 300x200-Pixel großes Fenster, das zunächst einen roten Hintergrund hat, nach 4 Sekunden seine Hintergrundfarbe auf Grün wechselt, dann wieder auf Rot zurückschaltet und sich schließlich selbsttätig schließt.)

Zwei Aspekte sollen hier nochmals hervorgehoben werden: die ist-ein-Beziehung und der Vererbungsaspekt. Jedes `MemoFrame`-Objekt *ist ein* `Frame`-Objekt. An allen Programmstellen, an denen ein `Frame`-Objekt stehen kann, kann auch ein `MemoFrame`-Objekt stehen; insbesondere kann ein `MemoFrame`-Objekt einer Variablen vom Typ `Frame` zugewiesen werden. Diese Eigenschaften fasst man zusammen, wenn man sagt, dass `MemoFrame` ein Subtyp von `Frame` ist. Ebenso ist `Student` ein Subtyp von `Druckbar`: Jedes `Student`-Objekt *ist ein* `Druckbar`-Objekt. An allen Programmstellen, an denen ein `Druckbar`-Objekt stehen kann, kann auch ein `Student`-Objekt stehen. Diese Subtypbeziehung ist im Prinzip unabhängig von Vererbungsaspekten.

Betrachten wir beide Beispiele in Bezug auf Vererbung, so ergibt sich folgender grundlegende Unterschied. Der Typ `Druckbar` hat keine „eigene“ Implementierung, sodass die Implementierung von `Student` keine Implementierungsteile von `Druckbar` erben kann. Ganz anders ist es beim zweiten Beispiel: `MemoFrame` erbt, erweitert und modifiziert eine sehr komplexe Funktionalität, in der insbesondere der Anschluss an das Fenstersystem enthalten ist. Die Modifikation ererbter Methoden wird dabei erreicht, indem eine Methode gleichen Namens definiert wird (im Beispiel: `setBackground`); man spricht vom „Überschreiben“<sup>4</sup> der Methode der Superklasse. Im Rumpf der neu definierten Methode kann mittels des Schlüsselworts `super` die ererbte Methode aufgerufen werden. Auf diese Weise ist die Anpassung und Erweiterung wiederverwendeter Programmteile möglich, ohne ihren Programmtext ändern zu müssen.

---

<sup>4</sup>Auf das Überschreiben von Methoden wird in Abschnitt 3.3.1.1 ausführlich eingegangen.

**Zusammenfassung.** Jedes Objekt hat eine klar definierte Schnittstelle bestehend aus seinen Methoden und (wie wir noch sehen werden) seinen Attributen. Diese saubere Schnittstellenbildung ermöglicht die Klassifikation von Objekten. Allgemeinere Objekte haben dabei eine kleinere Schnittstelle als die spezielleren Objekte. Bei der Bildung von Klassifikationen stehen zwei Techniken im Vordergrund: Mittels Abstraktion kann man gemeinsame Eigenschaften unterschiedlicher Typen zusammenfassen (Verkleinern der Schnittstelle). Mittels Spezialisierung kann man Typen und ihre Implementierung erweitern. Spezialisierung geht sehr häufig Hand in Hand mit dem Vererben von Implementierungsteilen und bekommt damit große Bedeutung für die Software-Wiederverwendung. (Dies führt oft dazu, dass „Vererbung“ als Oberbegriff für alle in diesem Abschnitt skizzierten Konzepte verwendet wird; insbesondere spricht man auch von der Vererbung von Schnittstellen.)

Die folgenden Abschnitte präzisieren die hier eingeführten Konzepte und erläutern deren Realisierung in Java.

## 3.2 Subtyping und Schnittstellen

Dieser Abschnitt erläutert zunächst den Zusammenhang zwischen Klassifikation und Typisierung. Er erklärt, was Subtyping bedeutet und wie es in Java realisiert ist. Dabei spielt der Begriff der Schnittstelle und das entsprechende Sprachkonstrukt von Java eine zentrale Rolle. Darüber hinaus analysiert der Abschnitt die Beziehung zwischen einem Typ und seinen Subtypen und skizziert den Zusammenhang zwischen Subtyping und parametrischen Typen. Schließlich demonstriert er die Anwendung von Subtyping im Rahmen verschiedener Programmier Techniken.

### 3.2.1 Subtyping und Realisierung von Klassifikationen

Wie wir bereits in den vorangegangenen Kapiteln gesehen haben, beschreibt ein Typ bestimmte Eigenschaften der ihm zugeordneten Werte und Objekte (vgl. die Abschnitte 1.3.1, S. 28 und 2.1.6, S. 100). Bisher sind wir dabei stillschweigend davon ausgegangen, dass jeder Wert und jedes Objekt zu genau einem Typ gehören. Andersherum ausgedrückt heißt das, dass es kein Objekt gibt, das zu zwei oder mehr Typen gehört. Diese Einschränkung ist sehr hinderlich, wenn man Klassifikationen programmtechnisch umsetzen will. Der einfachste Weg für eine Umsetzung wäre nämlich, jedem Begriff in der Klassifikation genau einen Typ zuzuordnen. Um die ist-ein-Beziehung widerzuspiegeln, müssten dann allerdings die Objekte der spezielleren Typen gleichzeitig zu den allgemeineren Typen gehören. Und dies würde die Einschränkung verletzen.

Da es ein zentrales Ziel der objektorientierten Programmierung ist, Klassifikationen zu unterstützen, gibt man in typisierten<sup>5</sup> objektorientierten Programmiersprachen die erläuterte Einschränkung auf und führt eine (partielle) Ordnung<sup>6</sup> auf der Menge der Typen ein, die sogenannte Subtyp-Ordnung: Wenn  $S$  ein Subtyp von  $T$  ist, wir schreiben dafür  $S \preceq T$ , dann gehören alle Objekte vom Typ  $S$  auch zum Typ  $T$ . Für  $S \preceq T$  und  $S \neq T$  schreiben wir auch einfach  $S \prec T$ . Diese Subtyp-Ordnung wird verwendet, um die strengen Typregeln von Sprachen ohne Subtyping zu liberalisieren. Insbesondere verlangt man bei der Zuweisung nicht mehr, dass der Typ der rechten Seite gleich dem Typ der linken Seite ist, sondern begnügt sich damit zu fordern, dass der Typ der rechten Seite ein Subtyp des Typs der linken Seite ist. Durch Verallgemeinerung erhält man die Grundregel für die Typisierung im Zusammenhang mit Subtyping:

*An allen Programmstellen, an denen ein Objekt vom Typ  $T$  zulässig ist, sind auch Objekte der Subtypen von  $T$  erlaubt.*

Eine Programmiersprache unterstützt *Subtyping*, wenn sie es ermöglicht, eine Subtyp-Ordnung auf ihren Typen zu definieren, und ihre Typregeln obiger Grundregel folgen.

Der Rest dieses Abschnitts besteht aus drei Teilen. Der erste Teil führt sogenannte Schnittstellentypen ein und erläutert, wie Subtyping in Java deklariert wird. Der zweite Teil behandelt den Zusammenhang zwischen Subtyping und Klassifikationen. Der dritte Teil beschreibt den Mechanismus zur Auswahl von Methoden.

### 3.2.1.1 Deklaration von Schnittstellentypen und Subtyping

In Java gibt es drei Möglichkeiten, Typen zu deklarieren: durch eine Klassendeklaration, durch eine *Schnittstellendeklaration* oder durch Deklaration von Feldern. Klassen deklarieren *Klassentypen* (vgl. Abschn. 2.1.2, S. 78) und Schnittstellen deklarieren *Schnittstellentypen*. Klassen-, Schnittstellen- und Feldtypen (vgl. Abschn. 1.3.1.2, S. 28) werden zusammenfassend als *Referenz- oder Objekttypen* bezeichnet. Außer den Referenztypen gibt es in Java die vordefinierten Basisdatentypen.

*Schnittstellendeklaration*

*Schnittstellentyp*

*Referenztyp*

Klassen haben wir in Kap. 2 behandelt. Eine Klasse deklariert einen neuen Typ zusammen mit dessen Implementierung. Eine Schnittstelle deklariert ebenfalls einen neuen Typ, legt aber nur die öffentliche Schnittstelle des Typs

<sup>5</sup>In untypisierten Sprachen taucht dieses Problem naturgemäß in dieser konkreten Form nicht auf. Wie wir weiter unten sehen werden, ist das Konzept, das Subtyping zugrunde liegt, aber allgemeiner und die allgemeinere Problematik gilt auch für untypisierte Sprachen.

<sup>6</sup>Eine binäre/zweistellige Relation  $\preceq$  auf einer Menge  $\mathcal{M}$  heißt *partielle Ordnung*, wenn sie reflexiv, antisymmetrisch und transitiv ist, d.h. wenn für alle Elemente  $S, T, U \in \mathcal{M}$  gilt:  $T \preceq T$  (Reflexivität); wenn  $S \preceq T$  und  $T \preceq S$  gilt, dann gilt  $S = T$  (Antisymmetrie); wenn  $S \preceq T$  und  $T \preceq U$  gilt, dann gilt auch  $S \preceq U$  (Transitivität).



fest. Ein Schnittstellentyp besitzt keine „eigene“ Implementierung und damit auch keine „eigenen“ Objekte. Die Objekte eines Schnittstellentyps sind die Objekte von dessen Subtypen. Ein erstes Beispiel dafür haben wir im Zusammenhang mit dem Schnittstellentyp `Druckbar` kennen gelernt (vgl. Abschn. 3.1, S. 150): `Student` ist ein Subtyp von `Druckbar`; die Implementierung von `Student` ist eine der Implementierungen des Typs `Druckbar`.

Dieser Unterabschnitt beschreibt, wie die Deklarationen von Schnittstellentypen in Java aussehen und wie deklariert wird, dass ein Typ ein Subtyp eines anderen ist.

**Deklaration von Schnittstellentypen.** Die Deklaration eines Schnittstellentyps legt den Typnamen, die Namen benannter Konstanten und die erweiterten Signaturen der Methoden fest, die der Typ zur Verfügung stellt. Da die Konstanten und Methoden immer öffentlich sind, kann und sollte der Modifikator `public` bei ihrer Deklaration unterdrückt werden. Schnittstellendeklarationen werden durch das Schlüsselwort `interface` kenntlich gemacht. Sie haben in der Praxis folgende syntaktische Form, wobei optionale Teile in das Klammerpaar `[ und ]opt` eingeschlossen sind:

```
[public]opt interface Schnittstellename
[ extends Liste von Schnittstellennamen ]opt
{
    Liste von Konstantendeklarationen und Methodensignaturen
}
```

Ein Schnittstellentyp kann öffentlichen oder paketlokalen Zugriff gewähren; im zweiten Fall kann der Typname nur innerhalb des Pakets verwendet werden, in dem die Deklaration steht.

Bevor wir die möglicherweise vorhandene Liste von Schnittstellennamen hinter dem Schlüsselwort `extends` erläutern, illustrieren wir die Deklaration von Schnittstellentypen mit Methoden anhand eines einfachen Beispiels:

```
interface Person {
    String getName();
    int getGeburtsdatum();
    void drucken();
    boolean hat_geburtstag( int datum );
}
```

Diese Schnittstellendeklaration führt den neuen Typ `Person` ein, der Methoden zur Verfügung stellt, um den Namen und das Geburtsdatum der Person abzufragen, um die Daten der Person zu drucken und um abzufragen, ob die Person an einem angegebenen Datum Geburtstag hat.

Wir wollen hier auch noch auf eine Besonderheit von Schnittstellendeklarationen im Gegensatz zu Klassendeklarationen hinweisen. Während es im

Kontext von Klassendeklarationen erlaubt ist, Attribute zu deklarieren, die zur öffentlichen Schnittstelle der Klasse gehören, kann man bei Schnittstellentypen zusätzlich zu Methoden lediglich benannte Konstanten deklarieren. Benannte Konstanten werden wie Attribute deklariert, die als öffentlich, statisch und unveränderlich vereinbart sind. Die Modifier `public`, `static` und `final` können und sollten in der Deklaration entfallen (vgl. Abschn. 2.1.4, S. 89). Benannte Konstanten wurden bis zur Java-Version 1.4 häufig – wie im folgenden Beispiel – zur Realisierung von Aufzählungstypen verwendet:

```
interface Farben {
    byte rot    = 0;
    byte gruen  = 1;
    byte blau   = 2;
    byte gelb   = 3;
}
```

Ab der Version 5.0 bietet Java eigene Aufzählungstypen an. Diese Art von Typen werden wir in Abschnitt 3.2.6.2 beschreiben.

**Deklaration von Subtyping.** Nachdem wir gesehen haben, wie Typen in Java deklariert werden, stellt sich die Frage, wann ein Typ  $S$  Subtyp eines Typen  $T$  ist. Bis zu einem gewissen Grade könnte man eine Subtyp-Beziehung automatisch aus den öffentlichen Schnittstellen, die die Typen bereitstellen, ableiten. In Java wird die Subtyp-Beziehung aber nicht automatisch ermittelt. Der Programmierer muss bei der Deklaration eines Typs  $S$  angeben, von welchen Typen  $S$  ein Subtyp sein soll. Der obige Schnittstellentyp `Person` ist also kein Subtyp von `Druckbar` (vgl. S. 150), auch wenn er u.a. die Methode `drucken` in seiner Schnittstelle anbietet.

Um die Sprechweisen zu vereinfachen, nennen wir  $T$  einen *Supertyp* von  $S$ , wenn  $S$  ein Subtyp von  $T$  ist. Wir sagen  $S$  ist ein *direkter* Subtyp von  $T$ , wenn  $S \prec T$  und wenn es keinen Typ  $U$  zwischen  $S$  und  $T$  gibt; d.h. es gibt kein  $U$  mit  $S \prec U$  und  $U \prec T$ . Entsprechend definieren wir, was ein direkter Supertyp ist. In Java wird bei jeder Deklaration eines Typs  $T$  angegeben, was die direkten Supertypen von  $T$  sein sollen. Fehlt die Angabe von Supertypen, wird der vordefinierte Typ `Object` als direkter Supertyp angenommen. Beispielsweise ist `Object` der einzige Supertyp des obigen Schnittstellentyps `Person`.

*Supertyp*

Ein Schnittstellentyp darf explizit nur Schnittstellentypen als Supertypen deklarieren; implizit ist darüber hinaus der Klassentyp `Object` ein Supertyp aller Schnittstellentypen. Als Beispiel deklarieren wir den Schnittstellentyp `Angestellte` mit direkten Supertypen `Person` und `Druckbar`:

```

interface Angestellte extends Person, Druckbar {
    String getName();
    int    getGeburtsdatum();
    int    getEinstellungsdatum();
    String getGehaltsklasse();
    void   drucken();
    boolean hat_geburtstag( int datum );
}

```

Da ein Subtyp-Objekt an allen Programmstellen erlaubt ist, an denen Objekte von seinem Supertyp zulässig sind, muss das Subtyp-Objekt auch mindestens die Methoden aller Supertypen besitzen. Selbstverständlich kann der Subtyp mehr Methoden anbieten (im Beispiel die Methoden `getEinstellungsdatum` und `getGehaltsklasse`). Da es lästig ist, die Methodensignaturen der Supertypen in den Subtypen wiederholen zu müssen, sieht Java auch bei Schnittstellen Vererbung vor: Die Signaturen und Attribute aller Supertypen werden automatisch an den neu deklarierten Typ vererbt. Statt der länglichen Deklaration von `Angestellte` hätte es demgemäß gereicht zu schreiben:

```

interface Angestellte extends Person, Druckbar {
    int    getEinstellungsdatum();
    String getGehaltsklasse();
}

```

Dieser Vererbungsmechanismus erklärt auch die Wahl des Schlüsselworts `extends`: Der Subtyp erweitert die explizit angegebene Schnittstelle.

Ein Klassentyp  $S$  darf als direkte Supertypen eine beliebige Anzahl von Schnittstellentypen  $T_1, \dots, T_n$  besitzen, aber nur einen Klassentyp  $T$ , die sogenannte *Superklasse* von  $S$ . Von der Superklasse wird die Implementierung geerbt. Für alle Methoden der Typen  $T_1, \dots, T_n$  muss  $S$  Implementierungen liefern. Syntaktisch wird die Superklasse durch das Schlüsselwort `extends` gekennzeichnet, die Schnittstellentypen durch das Schlüsselwort `implements`; insgesamt hat eine Klassendeklaration damit folgende Form, wobei die Modifikatorenliste leer sein kann:

```

Modifikatorenliste class Klassenname
[ extends      Klassenname ]opt
[ implements  Liste von Schnittstellentypnamen ]opt
{
    Liste von Attribut-, Konstruktor- und Methodendeklarationen
}

```

Ist keine Superklasse angegeben, ist die vordefinierte Klasse `Object` die direkte Superklasse. Was eine Klasse von ihrer Superklasse erbt und wie dieser Mechanismus genau funktioniert, wird in Abschn. 3.3 behandelt. Hier illustrieren wir zunächst, was es bedeutet, dass eine Klasse Schnittstellentypen

implementiert. Dazu betrachten wir die folgende Klasse `Student`, die die obigen Schnittstellen `Person` und `Druckbar` implementiert:

```
class Student implements Person, Druckbar {
    private String name;
    private int    geburtsdatum; /* in der Form JJJJMMTT */
    private int    matrikelnr;
    private int    semester;

    public Student( String n, int gd, int mnr, int sem ) {
        name = n;
        geburtsdatum = gd;
        matrikelnr = mnr;
        semester = sem;
    }

    public String  getName()           { return name; }
    public int     getGeburtsdatum()   { return geburtsdatum; }
    public int     getMatrikelnr()     { return matrikelnr; }
    public int     getSemester()       { return semester; }
    public void    drucken() {
        System.out.println("Name: " + name);
        System.out.println("Geburtsdatum: " + geburtsdatum);
        System.out.println("Matrikelnr: " + matrikelnr );
        System.out.println("Semesterzahl: " + semester );
    }

    public boolean hat_geburtstag ( int datum ) {
        return (geburtsdatum%10000) == (datum%10000);
    }
}
```

Die Klasse `Student` liefert Implementierungen, d.h. Methodenrümpfe, für alle Methoden der Schnittstellen `Person` und `Druckbar`. Kommt eine Methode in mehreren Schnittstellen mit der gleichen Signatur vor, wie z.B. die Methode `drucken`, wird sie im Subtyp als eine Methode betrachtet und muss deshalb auch nur einmal implementiert werden.

Es lohnt sich, die obige Fassung der Klasse `Student` mit derjenigen aus Unterabschn. 1.3.2.2, S. 44, zu vergleichen. Abgesehen von den unterschiedlichen Zugriffsrechten, die die Klassen gewähren, bieten ihre Objekte ein beinahe äquivalentes Verhalten. Beide Klassen besitzen Methoden mit der gleichen Funktionalität und sind Subtyp des entsprechenden Typs `Person`. Im hiesigen Fall wurde dies durch Implementierung eines Schnittstellentyps erreicht. In Kap. 1 wurde mit Vererbung gearbeitet. Die aus diesen Varianten resultierenden unterschiedlichen Möglichkeiten zur Realisierung von Typhierarchien werden wir in Abschnitt 3.2.1.2 vertiefen.

**Zusammenfassung.** Zusammenfassend lässt sich Folgendes feststellen: Die Subtyp-Beziehung muss in Java explizit deklariert werden. Dazu wird bei der Deklaration eines neuen Typs *T* angegeben, was die direkten Supertypen von *T* sein sollen. Bei Schnittstellentypen können als direkte Supertypen nur andere Schnittstellentypen angegeben werden. Bei Klassentypen gibt es genau

eine Superklasse – ist keine Klasse angegeben, ist `Object` die Superklasse; alle anderen direkten Supertypen müssen Schnittstellentypen sein. Alle Supertypen von  $T$  erhält man dann, indem man auch die Supertypen der direkten Supertypen von  $T$  hinzunimmt usw. (Bildung der transitiven Hülle; vgl. die Definition von partieller Ordnung auf S. 157).

### 3.2.1.2 Klassifikation und Subtyping

Subtyping ist ein hervorragendes Mittel zur programmtechnischen Realisierung von Klassifikationen. Jedem Begriff in der Klassifikation wird dabei ein Typ zugeordnet; die Beziehung der Begriffe in der Klassifikation wird durch die Subtyp-Beziehung modelliert. Da ein Typ mehrere direkte Supertypen besitzen kann, braucht die Klassifikation auch nicht streng hierarchisch/baumartig zu sein. Wie beim Quadrat in der Klassifikation geometrischer Figuren in Abb. 3.1, S. 148, kann ein Begriff durchaus mehrere Verallgemeinerungen besitzen.

Bei der Umsetzung einer Klassifikation muss man sich überlegen, welche Begriffe man als Klassentypen und welche als Schnittstellentypen realisieren möchte. Zwei unterschiedliche Realisierungsmöglichkeiten wollen wir am Beispiel von Abb. 3.3 diskutieren. Jeder der angegebenen Namen in der Abbildung ist ein Typname. Die Pfeile veranschaulichen die Subtyp-Ordnung (`Druckbar` ist ein Subtyp von `Object`; `WissAngestellte` ist ein Subtyp von `Angestellte`, `Druckbar`, `Person` und `Object`; usw.).

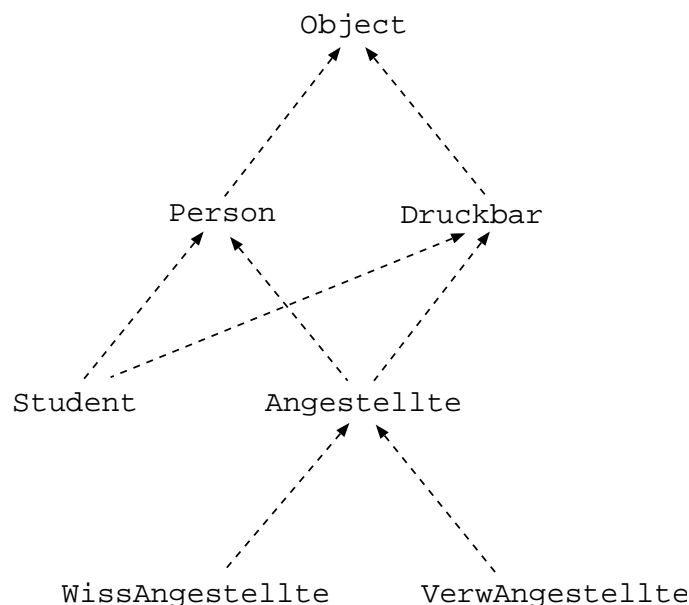


Abbildung 3.3: Subtyp-Beziehungen

**Erste Realisierung.** Als erste Realisierungsmöglichkeit betrachten wir eine Implementierung, in der nur diejenigen Typen als Klassentypen deklariert werden, die keine Subtypen besitzen, also die Typen `Student`, `WissAngestellte` und `VerwAngestellte`. Für die Typen `Druckbar`, `Person` und `Angestellte` benutzen wir die obigen Schnittstellendeklarationen (vgl. die Seiten 150, 158 und 160); der Typ `Object` ist in Java vordefiniert. Eine Klassendeklaration für `Student` haben wir auf S. 161 behandelt; Klassendeklarationen für `WissAngestellte` und `VerwAngestellte` könnten wie in Abb. 3.4 aussehen.

```
class WissAngestellte implements Angestellte {
    private String name;
    private int    geburtsdatum; /* in der Form JJJJMMTT */
    private int    einstellungsdatum;
    private String gehaltsklasse;
    private String fachbereich;
    private String lehrgebiet;

    public WissAngestellte( ... ) { ... }

    public String  getName() { return name; }
    public int     getGeburtsdatum() { return geburtsdatum; }
    public int     getEinstellungsdatum() { ... }
    public String  getGehaltsklasse() { ... }
    public String  getFachbereich() { ... }
    public String  getLehrgebiet() { ... }
    public void    drucken() { ... }
    public boolean hat_geburtstag( int datum ) { ... }
}

class VerwAngestellte implements Angestellte {
    private String name;
    private int    geburtsdatum; /* in der Form JJJJMMTT */
    private int    einstellungsdatum;
    private String gehaltsklasse;
    private int    dezernat;

    public VerwAngestellte( ... ) { ... }

    public String  getName() { return name; }
    public int     getGeburtsdatum() { return geburtsdatum; }
    public int     getEinstellungsdatum() { ... }
    public String  getGehaltsklasse() { ... }
    public int     getDezernat() { ... }
    public void    drucken() { ... }
    public boolean hat_geburtstag( int datum ) { ... }
}
```

Abbildung 3.4: Die Klassen `WissAngestellte` und `VerwAngestellte`

Die Klassen `Student`, `WissAngestellte` und `VerwAngestellte` implementieren alle die Schnittstellen `Druckbar` und `Person`. Die Klassen `WissAngestellte` und `VerwAngestellte` implementieren darüber hinaus die Schnittstelle `Angestellte`.

Bei dieser Realisierung lassen sich nur Objekte zu den Typen `Student`, `WissAngestellte` und `VerwAngestellte` erzeugen. Diese Objekte gehören aber gleichzeitig zu den entsprechenden Supertypen. So gehört beispielsweise ein `Student`-Objekt zu den Typen `Druckbar`, `Person` und `Object`. Andersherum betrachtet gehört jedes Objekt eines Schnittstellentyps zu einem der Subtypen. Insbesondere gehört jedes Objekt vom Typ `Angestellte` entweder zur Klasse `WissAngestellte` oder zur Klasse `VerwAngestellte`.

**Zweite Realisierung.** Als alternative Realisierung versuchen wir, möglichst viele der Typen als Klassen zu implementieren. Da Java es nicht gestattet, dass eine Klasse mehrere Superklassen hat, können wir allerdings nicht gleichzeitig `Person` und `Druckbar` als Klasse realisieren (andernfalls hätte beispielsweise `Student` zwei Superklassen). Wir entscheiden uns dafür, `Person` als Klasse und `Druckbar` als Schnittstelle zu deklarieren; denn beim Typ `Druckbar` ist unklar, wie seine Implementierung aussehen sollte. Das Einzige, was wir über diesen Typ wissen, ist, dass seine Objekte eine Methode `drucken` bereitstellen; wie die Objekte aussehen und was die Methode im Einzelnen ausgeben soll, ist unspezifiziert.

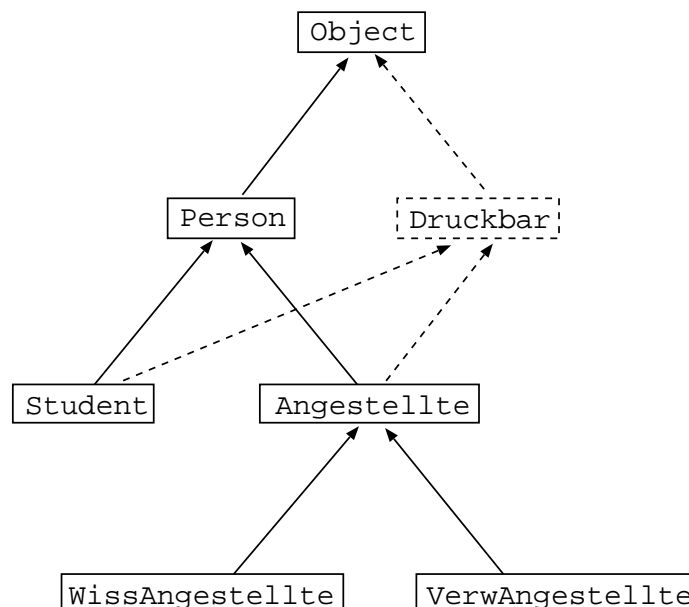


Abbildung 3.5: Subtyping realisiert durch Klassen und Schnittstellen

Eine Übersicht über diese Realisierung bietet Abb. 3.5. Sie verfeinert

Abb. 3.3. Für jeden Typ ist angegeben, ob er als Klasse oder als Schnittstelle realisiert ist: Durchgezogene Rahmen markieren Klassen, gestrichelte Rahmen Schnittstellen. Außerdem sind die Pfeile zwischen Super- und Subklassen durchgezogen, um hervorzuheben, an welchen Stellen Subtyping mit Vererbung von Implementierungsteilen gekoppelt ist.

Auf Implementierungen für die Klassen dieser Realisierungsvariante werden wir im Zusammenhang mit Vererbung in Abschn. 3.3 eingehen (vgl. auch Abschn. 1.3.2, S. 48). Zur Einübung der Java-Syntax zeigen wir hier nur die Köpfe der Typdeklarationen:

```
class Person { ... }  
interface Druckbar { ... }  
class Student extends Person implements Druckbar { ... }  
class Angestellte extends Person implements Druckbar { ... }  
class WissAngestellte extends Angestellte { ... }  
class VerwAngestellte extends Angestellte { ... }
```

Abbildung 3.6: Die zu Abbildung 3.5 passende Realisierung

Bei dieser zweiten Variante gibt es auch Objekte, die vom Typ `Person` bzw. vom Typ `Angestellte` sind, ohne zu einem der Subtypen zu gehören. Bei der Realisierung von Klassifikationen ist dies in vielen Fällen durchaus natürlich und erwünscht. Am Beispiel der geometrischen Figuren von Abb. 3.1, S. 148, lässt sich das gut demonstrieren: Es gibt Parallelogramme, die weder Rauten noch Rechtecke sind; es gibt Polygone, die weder Dreiecke noch Vierecke sind. Analoges könnte man in obigem Beispiel anführen. Eine Person kann ein verbeamteter Professor sein. Der ist weder ein Student noch ein Angestellter.

**Klassen- vs. Schnittstellentypen.** Die beiden Realisierungsvarianten geben bereits einige Hinweise darauf, in welchen Fällen es sinnvoll ist, einen Typ als Schnittstelle umzusetzen, und in welchen Fällen eine Klassendeklaration angebracht ist.

Wenn wir außer der Methoden-Schnittstelle nur sehr wenig über einen Typ wissen oder uns auf keine Implementierungsaspekte festlegen wollen, bietet es sich an, einen Typ als Schnittstelle zu realisieren. Wie wir sehen werden, kommt dies vor allem am oberen Ende von Typhierarchien vor, d.h. bei den allgemeinen, abstrakten Typen (ein gutes Beispiel, um dies zu studieren, bieten die Strom-Typen in der Java-Bibliothek; vgl. Abschn. 4.3). Da Java die Verwendung mehrerer Superklassen nicht gestattet, müssen darüber hinaus Schnittstellentypen verwendet werden, wenn eine Klasse mehrere Supertypen haben soll.

Eine Klassendeklaration wird man zur Realisierung eines Typs verwenden, wenn der Typ auch ohne Subtypen angewendet werden soll (siehe



die Diskussion von Parallelogrammen im letzten Absatz) oder wenn der Typ mit einer Implementierung versehen werden soll. Die Implementierung kann dann in Subklassen spezialisiert werden. Durch Vererbung von Implementierungsteilen kann dabei viel an Programmierarbeit und Programmgröße gespart werden. Bereits anhand der kleinen Klassen `Student`, `WissAngestellte` und `VerwAngestellte` sieht man, wie fast gleichlautender Programmtext innerhalb einer Typhierarchie immer wieder vorkommt. Dies kann vermieden werden, wenn man Typen, wie `Person` und `Angestellte`, als Klassen auslegt und Vererbung verwendet.

*abstrakte  
Klassen*

Im Übrigen bietet Java eine Zwischenform zwischen Schnittstellentypen und Klassentypen an: die sogenannten *abstrakten Klassen*. Sie erlauben es, Typen mit unvollständigen Implementierungen zu deklarieren. Eine abstrakte Klasse ähnelt einem Schnittstellentyp darin, dass es (wegen der Unvollständigkeit der Implementierung) nicht möglich ist, Objekte dieser Klasse zu erzeugen (s. auch Seite 158). Eine abstrakte Klasse ähnelt einer normalen Klasse darin, dass die Implementierungsteile an Subklassen vererbt werden können. (Abstrakte Klassen werden genauer in Abschn. 3.3 erläutert.)

**Typhierarchien erweitern.** In den meisten Fällen wird man beim Entwickeln eines objektorientierten Programms existierende Typhierarchien verwenden und ggf. erweitern. Das Erweitern einer Typhierarchie „nach unten“, d.h. das Hinzufügen speziellerer Typen, ist in Java ohne Änderung der existierenden Typdeklarationen möglich: Die hinzugefügte Typdeklaration legt fest, was ihre Supertypen sind, und erweitert so auch die Subtyp-Ordnung. Beispielsweise könnten wir der obigen Typhierarchie folgende Klasse `Firma` hinzufügen, ohne dass Änderungen an den anderen Typdeklarationen notwendig werden:

```
class Firma implements Druckbar {
    private String name;
    ...
    public String getName() { return name; }
    public void drucken() { ... }
}
```

Aufwendiger ist es in Java, einen neuen Typen hinzuzufügen, der existierende Typen abstrahieren soll. Beispielsweise könnten wir bei der Benutzung der um `Firma` erweiterten Typhierarchie feststellen, dass es hilfreich wäre, einen Typ für alle diejenigen Objekte zur Verfügung zu haben, die druckbar sind und einen Namen haben, also folgende Schnittstelle implementieren:

```
interface DruckbarBenannt extends Druckbar {
    String getName();
}
```

Dieser Typ soll als Abstraktion der Schnittstelle `Person` und der Klasse `Firma` in die Typhierarchie eingefügt werden. Das geht nicht ohne Änderung der Köpfe von `Person` und `Firma`:

```
interface Person extends DruckbarBenannt { ... }
class Firma implements DruckbarBenannt { ... }
```

Insgesamt lässt sich sagen, dass Java das Spezialisieren einer Typhierarchie gut unterstützt, während das Abstrahieren Änderungen am existierenden Programmtext notwendig macht. Dass das Abstrahieren Programmänderungen notwendig macht, hängt mit der eingeschränkten Art zusammen, wie die Subtyp-Beziehung in Java deklariert wird. Im Gegensatz dazu erlaubt es die Programmiersprache Sather beispielsweise auch, einen Typ als Supertyp existierender Typen zu deklarieren, so dass Spezialisierung und Abstraktion in ähnlicher Weise unterstützt werden (vgl. [SOM94]).

### 3.2.1.3 Subtyping und dynamische Methodenauswahl

Bisher haben wir die Programmkonstrukte kennen gelernt, mit denen Subtyping deklariert wird, und haben gesehen, wie man Subtyping nutzen kann, um Klassifikationen zu realisieren. In diesem Unterabschnitt wollen wir uns näher anschauen, was Subtyping im Zusammenhang mit der Anwendung von Methoden bedeutet. Die Stärke von Subtyping kommt nämlich erst dadurch richtig zum Tragen, dass Algorithmen über einem Typ  $T$  so formuliert werden können, dass sie auch für alle Subtypen von  $T$  funktionieren.

**Dynamische Methodenauswahl.** Entsprechend der Grundregel für Subtyping kann eine Variable bzw. ein Parameter von einem Typ  $T$  Objekte der Subtypen von  $T$  referenzieren. Eine Variable vom Typ `Druckbar` kann also insbesondere `Student`- und `Firma`-Objekte referenzieren. Die Variable `e` in der Klasse `Behaelter` auf S. 151 lieferte dafür ein erstes Beispiel. Ein ähnlich gelagertes Szenario zeigt die folgende Klasse:

```
class DruckMitLaufNr {
    private static int laufendeNr = 1;
    public static void nrdrucken( Druckbar db ) {
        System.out.println("LfdNr. " + laufendeNr + ":");
        (*) db.drucken();
        laufendeNr++;
    }
}
```

Sie formuliert unter Benutzung des Typs `Druckbar` einen neuen (trivialen) „Algorithmus“: Ihre Methode `nrdrucken` nimmt ein druckbares Objekt als Parameter, gibt eine laufende Nummer aus, druckt die Informationen zum

übergebenen Parameter und inkrementiert die laufende Nummer. Dazu wird in der mit (\*) markierten Zeile die „Methode“ `drucken` der Schnittstelle `Druckbar` auf dem Parameter `db` aufgerufen. In der Schnittstelle `Druckbar` ist für diese „Methode“ aber keine Implementierung angegeben. Implementierungen gibt es nur in den Subtypen von `Druckbar`.

Gemäß dem objektorientierten Grundmodell ist jeweils die Methode des Objekts auszuwählen, das von `db` referenziert wird: Für ein `Student`-Objekt ist also die Methode `drucken` der Klasse `Student` auszuführen; entsprechend ist für ein `Firma`-Objekt dessen `Druckmethode` auszuwählen, etc. Da das von `db` referenzierte Objekt von Aufruf zu Aufruf der Methode `nrdrucken` unterschiedlich sein kann, kommen im Allgemeinen in der mit (\*) markierten Zeile verschiedene Methoden zur Ausführung.

Zur Übersetzungszeit kann also für die angegebene Programmstelle nicht festgelegt werden, welcher Programmcode auszuführen ist. Dies kann erst zur Laufzeit des Programms in Abhängigkeit vom aktuellen Parameter `db` bestimmt werden. Deshalb spricht man von *dynamischer Methodenauswahl* oder *dynamischem Binden*. Im Englischen werden dafür häufig die Begriffe *dynamic binding* oder *dynamic dispatch* verwendet.

*dynamische  
Methoden-  
auswahl*

**Subtyping zusammengefasst.** Die kleine Beispielklasse `DruckMitLaufNr` zeigt alle prinzipiellen Aspekte von Subtyping und dynamischem Binden. Auch wenn man die Bedeutung, die diese Techniken für die Programmierung und Wiederverwendung von Programmen in der Praxis mit sich bringen, vermutlich erst angesichts größerer Beispiele ermessen kann, scheint es trotzdem sinnvoll, sich die prinzipiellen Aspekte einmal explizit zu vergegenwärtigen:

- Subtyping erlaubt es, gemeinsame Eigenschaften unterschiedlicher Typen in Form eines allgemeineren Typs auszudrücken.
- Algorithmen können auf der Basis von allgemeinen Typen formuliert werden. Sie brauchen also nicht für jeden der (spezielleren) Subtypen neu angegeben zu werden (die Methode `nrdrucken` konnte auf der Basis des Typs `Druckbar` formuliert werden, so dass es sich erübrigt, sie für jeden Subtyp von `Druckbar` anzugeben).
- Dynamisches Binden ist notwendig, um den allgemein formulierten Algorithmus auf die konkreten Objekte anwenden zu können.
- Algorithmen, die auf der Basis allgemeiner Typen formuliert sind, funktionieren auch für solche Subtypen, die dem Programm erst nachträglich hinzugefügt werden; Änderungen am existierenden Programmtext sind dafür nicht nötig (beispielsweise funktioniert die Methode `nrdrucken` auch für alle zukünftigen Subtypen von `Druckbar`).

Damit Subtyping wie beschrieben funktioniert, muss ein Subtyp bestimmten Bedingungen genügen. Insbesondere muss ein Subtyp alle Methoden des Supertyps besitzen. Diese Bedingungen werden im folgenden Abschnitt weiter untersucht.

### 3.2.2 Subtyping genauer betrachtet

Der letzte Abschnitt hat Subtyping eingeführt und gezeigt, wie man es nutzen kann, um Klassifikationen zu realisieren. Dieser Abschnitt wendet sich den mehr technischen Aspekten von Subtyping zu, wie sie in üblichen Programmiersprachen auftreten. Er geht zunächst auf die Subtyp-Beziehung zwischen vordefinierten und benutzerdefinierten Typen am Beispiel von Java ein. Dann untersucht er genauer, was erfüllt sein muss, damit ein Typ Subtyp eines anderen sein kann, und was die Subtyp-Beziehung darüber hinaus bedeuten kann. Schließlich erläutert er den Begriff Polymorphie und vergleicht die drei Formen der Polymorphie, die bisher behandelt wurden.

#### 3.2.2.1 Subtyping bei vordefinierten Typen und Feldtypen

In Unterabschn. 3.2.1.1 wurde beschrieben, wann ein Schnittstellen- bzw. Klassentyp Subtyp eines anderen Schnittstellen- bzw. Klassentyps ist. Bei den Schnittstellen- bzw. Klassentypen unterscheiden wir zwischen den *benutzerdefinierten* Typen und den Typen, die in der Java-Bibliothek definiert sind. Wie wir gleich sehen werden, kommt dabei dem im Paket `java.lang` definierten Typ `Object` eine Sonderrolle zu. Dieser Abschnitt geht der Frage nach, wie die Subtyp-Beziehung zwischen Schnittstellen-, Klassen-, Feldtypen und den in Kap. 1 erläuterten Basisdatentypen ist.

**Ein allgemeinsten Typ für Objekte.** Jeder benutzerdefinierte Typ  $T$  ist Subtyp des vordefinierten Typs `Object`. Dies ergibt sich im Wesentlichen aus zwei bereits erläuterten Festlegungen:

1. `Object` ist per default der Supertyp aller Typen, bei denen nicht explizit ein Supertyp deklariert ist (vgl. S. 159).
2. Die Subtyp-Beziehung darf keine Zyklen enthalten<sup>7</sup>, folgende Deklarationen sind also beispielsweise unzulässig:

```
interface A extends B { }
interface B extends C { }
interface C extends A { }
```

<sup>7</sup>Sonst wäre sie keine partielle Ordnung; im nachfolgenden Beispiel gilt nämlich  $A \preceq B$  und  $B \preceq A$ , also ist die Subtyp-Beziehung nicht antisymmetrisch.

Da nach der ersten Festlegung jeder benutzerdefinierte Typ mindestens einen Supertyp hat, es nur endlich viele Typen gibt und keine Zyklen existieren, kommt man nach endlich vielen Schritten zum Typ `Object`, wenn man von einem Typ jeweils zu einem seiner Supertypen geht (dabei stellt man sich die Subtyp-Beziehung am besten als einen nicht-zyklischen, gerichteten Graphen vor; vgl. z.B. Abb. 3.3).

Wie wir noch sehen werden, ist es sehr hilfreich, einen solchen allgemeinsten Typ für Objekte zu besitzen. Beispielsweise lassen sich damit Behälter realisieren, die beliebige Objekte als Elemente aufnehmen können. Noch günstiger ist es, wenn es einen Typ gibt, der sowohl Supertyp aller Objekttypen als auch aller Werttypen ist. Dies ist z.B. in der Sprache Smalltalk der Fall, in der die Werte der üblichen Basisdatentypen als konstante Objekte behandelt werden. In Java gibt es nur für Referenztypen einen allgemeinsten Typ, nämlich `Object`. `Object` ist aber kein Supertyp der Basisdatentypen. Ein Grund dafür besteht vermutlich darin, dass man die resultierenden Effizienz Nachteile vermeiden wollte.

**Subtyping bei Feldern.** Jeder Feldtyp ist ein Subtyp von `Object`. Eine Variable vom Typ `Object` kann also sowohl (Referenzen auf) Objekte beliebiger Klassentypen speichern, als auch (Referenzen auf) Felder. Darüber hinaus ist ein Feldtyp `CS[]` genau dann ein Subtyp eines anderen Feldtyps `CT[]`, wenn `CS` ein Subtyp von `CT` ist.

Die ersten beiden der folgenden Zuweisungen sind also zulässig; die dritte wird vom Übersetzer nicht akzeptiert (`Person` ist in dem Beispiel ein Klassentyp):

```
Object   ovar = new String[3];
Person[] pfv = new Student[2];
Student[] sfv = new Person[2]; // unzulässig
```

Die beschriebene Festlegung bzgl. der Subtyp-Beziehung zwischen Feldtypen ist in vielen praktischen Fällen sehr hilfreich. Sie birgt aber eine leicht zu übersehende Fehlerquelle, die als Einführung in die Problematik der Typsicherheit im Zusammenhang mit Subtyping sehr illustrativ ist. Zur Diskussion betrachten wir folgendes Beispiel:

```
(1) String[] strfeld = new String[2];
(2) Object[] objfeld = strfeld;
(3) objfeld[0] = new Object();           // Laufzeitfehler!!!
                                           // ArrayStoreException
(4) int strl   = strfeld[0].length();
```

Die Zuweisung in Zeile (2) ist korrekt, da `String` ein Subtyp von `Object` ist, also `String[]` ein Subtyp von `Object[]`. Solange man nur lesend auf

das von `objfeld` referenzierte Feld zugreift, entsteht aus der Zuweisung auch kein Problem, da jedes Objekt, das man aus dem Feld ausliest, vom Typ `String` ist, also auch zum Typ `Object` gehört. Anders ist es, wenn man wie in Zeile (3) eine Komponente des Felds verändert, beispielsweise indem man ihr ein neues Objekt der Klasse `Object` zuweist. Aus Sicht der Variablen `objfeld`, die eine Referenz vom Typ `Object[]` hält, ist nach wie vor alles in Ordnung. Aus Sicht der Variablen `strfeld`, die eine Referenz auf dasselbe Feldobjekt besitzt, stimmt die Welt aber nicht mehr: Die Benutzer der Variablen `strfeld` glauben eine Referenz auf ein Feld zu haben, in dem ausschließlich `String`-Objekte eingetragen sind (oder `null`). Nach der Zuweisung in Zeile (3) speichert die nullte Komponente aber eine Referenz auf ein Objekt der Klasse `Object`. Der scheinbar harmlose Aufruf der Methode `length` aus Klasse `String` in Zeile (4) würde damit zu einer undefinierten Situation führen: Objekte der Klasse `Object` besitzen keine Methode `length`.

Dies widerspricht einem der zentralen Ziele der Typisierung objektorientierter Programme. Die vom Übersetzer durchgeführte Typprüfung soll nämlich insbesondere garantieren, dass Objekte nur solche Nachrichten erhalten, für die sie auch eine passende Methode besitzen. Java löst die skizzierte Problematik durch einen Kompromiss. Um einerseits Zuweisungen wie in Zeile (2) zulassen zu können und andererseits undefinierte Methodenaufrufe zu vermeiden, „verbietet“ es Zuweisungen wie in Zeile (3). Da man ein solches Verbot im Allgemeinen aber nicht statisch vom Übersetzer abtesten kann, wird zur Laufzeit eine Ausnahme erzeugt, wenn einer Feldkomponente ein Objekt von einem unzulässigen Typ zugewiesen wird; d.h. eine solche Zuweisung terminiert dann abrupt (zur abrupten Terminierung und Ausnahmebehandlung vgl. Unterabschn. 1.3.1.3, S. 38).

**Basisdatentypen und Subtyping.** Zwischen den Basisdatentypen gibt es grundsätzlich keine Subtyp-Beziehung. Die in Abschn. 1.3.1 beschriebenen Möglichkeiten, zwischen diesen Typen zu konvertieren, bzw. die automatisch vom Übersetzer vorgenommenen Konvertierungen lassen allerdings die kleineren Zahlentypen wie Subtypen der größeren erscheinen (beispielsweise kann man `short` wie einen Subtyp von `int` behandeln).

Die Basisdatentypen in Java stehen auch in keiner Subtyp-Beziehung zu den Referenztypen; insbesondere sind die Basisdatentypen nicht Subtyp von `Object`. Dies erweist sich in vielen Situationen als echter programmtechnischer Nachteil. Benötigt man beispielsweise eine Liste, deren Elemente sowohl Zahlen als auch Objekte sein können, muss man die Zahlen in Objekte „verpacken“. Wir demonstrieren diese Lösung im folgenden Beispiel anhand der Klasse `Int`:

```

class Int {
    public int value;
    public Int( int i ) {
        value = i;
    }
}

class TestIntWrapper {
    public static void main( String[] argf ){
        LinkedList<Object> ll = new LinkedList<Object>();
        ll.addLast( new Student("Planck",18580423,3454545,47) );
        ll.addLast( new String("Noch ein Listenelement") );
        ll.addLast( new Int( 7 ) );
        (*) int i = ((Int) ll.getLast()).value;
    }
}

```

Um einen `int`-Wert in eine Liste mit Elementen vom Typ `Object` eintragen zu können, speichert man ihn im Attribut eines `Int`-Objekts. Dieses kann in die Liste eingetragen werden. Zum Auslesen eines Werts lässt man sich das entsprechende Listenelement geben, konvertiert dieses auf den (Sub-)Typ `Int` und greift auf das Attribut `value` zu.

Das Bibliothekspaket `java.lang` enthält für jeden Basisdatentypen eine entsprechende Klasse mit einem Attribut `value` zum Speichern der Werte des Basisdatentyps. Da diese Klassen die Werte quasi in Objekte einpacken, werden sie üblicherweise Wrapper-Klassen genannt. Anders als in der Klasse `Int` sind die `value`-Attribute in diesen Wrapper-Klassen als `private` deklariert. Für das Auslesen des gespeicherten Wertes stellen die Wrapper-Klassen deshalb geeignete Methoden zur Verfügung. Die Klasse `Integer` verwendet dafür z.B. die Methode `intValue()`. Im Vergleich zu einer Subtyp-Beziehung zwischen dem Typ `Object` und den Basisdatentypen haben Wrapper-Klassen den Nachteil, dass die Programme schlechter lesbar werden und mehr Speicherplatz verbrauchen. Andererseits würde die Einführung einer Subtyp-Beziehung zwischen `Object` und den Basisdatentypen eine Implementierung der Basisdatentypen erschweren und auch einen gewissen Effizienzverlust verursachen

Wrapper-  
Klassen

Autoboxing

Deshalb gibt es ab der Java Version 5.0 das sogenannte „*Autoboxing*“, das das automatische Verpacken von Werten in Wrapperobjekte sowie das automatische Entpacken aus Wrapperobjekten unterstützt. Dadurch wird die programmtechnische Handhabung von Wrappertypen wesentlich vereinfacht. Man unterscheidet beim Autoboxing das „*Boxing*“ und das „*Unboxing*“. *Boxing* konvertiert einen Basisdatentyp in seinen zugehörigen Wrappertyp und *Unboxing* konvertiert einen Wrappertyp in seinen zugehörigen Basisdatentyp.

Boxing

Unboxing

Kommt in einem Programm z.B. ein Ausdruck `e` vom Typ `int` vor, wo ein

Ausdruck vom Typ `Integer` erwartet wird, wird dieser Ausdruck automatisch in `new Integer(e)` konvertiert. Umgekehrt, kommt im Programm ein Ausdruck `e` vom Typ `Integer` vor, wo ein Ausdruck vom Typ `int` erwartet wird, wird `e` automatisch nach `e.intValue()` konvertiert. Das folgende, [NW07] entnommene Beispiel zeigt den Unterschied in der Handhabung zwischen der Java Version 5.0 und früheren Versionen:

```
// Version ab Java 5.0
List<Integer> ints = new ArrayList<Integer>();
(*) ints.add(1);
(**) int n = ints.get(0);

// Noetige explizite Verwendung von new Integer(...)
// und intValue zum Konvertieren zwischen
// Wrappervariante und Basisdatentyp in Versionen < 5.0
List<Integer> ints = new ArrayList<Integer>();
ints.add(new Integer(1));
int n = ints.get(0).intValue();
```

Zeile (\*) zeigt den Vorgang des Boxings: Die `add`-Methode erwartet einen Parameter vom Typ `Integer`. Tatsächlich übergeben wird aber der Wert 1. Dieser wird automatisch umgesetzt nach `new Integer(1)`. Zeile (\*\*) demonstriert den umgekehrten Fall. `n` ist eine Variable vom Typ `int`, die `get`-Methode liefert aber einen Wert vom Typ `Integer`, der jetzt automatisch in einen `int`-Wert ausgepackt wird.

### 3.2.2.2 Was es heißt, ein Subtyp zu sein

Die Grundregel für Subtyping besagt, dass ein Objekt von einem Subtyp an allen Programmstellen vorkommen kann, an denen Supertyp-Objekte zulässig sind. Dazu muss der Subtyp die Eigenschaften des Supertyps besitzen. Diese Eigenschaften betreffen sowohl syntaktische Bedingungen als auch ein konformes Verhalten des Subtyps. Dieser Unterabschnitt erläutert die syntaktischen Bedingungen und geht schließlich kurz auf konformes Verhalten ein.

**Syntaktische Bedingungen.** Damit Subtyping funktioniert, müssen Subtyp-Objekte alle Operationen unterstützen, die auch vom Supertyp angeboten werden. Insbesondere muss ein Subtyp-Objekt alle Methoden und Attribute des Supertyps besitzen. Betrachten wir dazu beispielsweise die folgende Klasse, die diese Regel verletzt (und darum vom Java-Übersetzer nicht akzeptiert wird):

```
class NichtDruckbar implements Druckbar {
    // keine Methode
}
```



Bei einem Aufruf der Methode `nrdrucken` (vgl. Klasse `DruckMitLaufNr` auf S. 167)

```
DruckMitLaufNr.nrdrucken( new NichtDruckbar() );
```

wäre unklar, was an der Programmstelle `db.drucken()` zu tun ist, da Objekte des Typs `NichtDruckbar` keine Methode `drucken` besitzen. Entsprechendes gilt für den Zugriff auf Attribute.

Es reicht aber nicht zu fordern, dass es in den Subtypen Attribute und Methoden gleichen Namens gibt. Die Attribute müssen auch in ihrem Typ, die Methoden in den Typen der Parameter, den Ausnahmetypen und dem Ergebnistyp zusammenpassen. In Java wird bei Versionen  $< 5.0$  verlangt, dass die Attribut-, Parameter- und Ergebnistypen in Super- und Subtyp gleich sein müssen. (Die Version 5.0 ermöglicht für Ergebnistypen jetzt schwächere Bedingungen, s.u.) Folgendes Beispiel ist also ein korrektes Java-Programm:

```
class ExceptionS extends ExceptionT { }

interface Supertyp {
    ReturnType meth( ParameterType p) throws ExceptionT;
}

class Subtyp implements Supertyp {
    public ReturnType meth( ParameterType p)
                               throws ExceptionS { ... }
}
```

Im Beispiel sind die Parameter- und Ergebnistypen der Methode `meth` in `Supertyp` und `Subtyp` gleich. Der Ausnahmetyp `ExceptionS` der Methodendeklaration in `Subtyp` ist allerdings nur ein Subtyp des Ausnahmetyps `ExceptionT`. Dies ist in Java zulässig. Im Prinzip könnte man auch bei den Parametertypen mehr Flexibilität erlauben. Entscheidend ist nur, dass die Methode des Subtyps in allen Programmkontexten korrekt anwendbar ist, in denen die Methode des Supertyps vorkommen kann; denn die dynamische Methodenauswahl führt dazu, dass die Subtyp-Methode anstatt der Supertyp-Methode ausgeführt werden kann.

Genauer gesagt muss Folgendes für zwei Typen *Sub* und *Super* mit  $Sub \preceq Super$  gelten:

- Die überschreibende Methode von *Sub* – hier als `methsub` bezeichnet – muss mindestens die aktuellen Parameter verkraften können, die die überschriebene Methode `methsuper` von *Super* verarbeiten kann; d.h. jeder Parametertyp von `methsub` muss ein *Supertyp* des entsprechenden Parametertyps von `methsuper` sein.

- Die Methode `methsub` darf nur Ergebnisse liefern, die auch als Ergebnisse von der Methode `methsuper` zulässig sind; d.h. der Ergebnistyp von `methsub` muss ein *Subtyp* des Ergebnistyps `methsuper` sein. Dies ist nötig, wenn das Ergebnis des Methodenaufrufs in einem Ausdruck ausgewertet wird, z.B. bei einer Zuweisung an eine Variable. Im Kontext von `methsuper` wird in einem solchen Fall mindestens dessen Ergebnistyp erwartet.

Da die Subtyp-Beziehung bei den Parametertypen gerade umgekehrt zur Subtyp-Beziehung auf den die Methoden `methsub` und `methsuper` deklarierenden Typen (hier *Sub* und *Super*) ist, spricht man von einer *kontravarianten* Beziehung oder kurz von *Kontravarianz*. Die Ergebnistypen und Ausnahmetypen verhalten sich entsprechend der Subtyp-Beziehung auf den die Methoden `methsub` und `methsuper` deklarierenden Typen – man spricht von einer *kovarianten* Beziehung oder kurz von *Kovarianz*.

Kontravarianz

Kovarianz

Java lässt Kontravarianz bei den Parametertypen nicht zu, da dies zu Problemen im Zusammenhang mit überladenen Methodennamen führen würde (vgl. Abschn. 2.1.4, S. 91, vgl. Abschn. 3.3.5, S. 251). Ab der Version 5.0 lässt Java aber Kovarianz bei den Ergebnistypen zu, was in früheren Versionen noch nicht erlaubt war.

**Konformes Verhalten.** In unserer Untersuchung von Subtyping haben wir uns bisher nur mit syntaktischen Bedingungen beschäftigt: Damit ein Typ als Subtyp eines anderen fungieren kann, muss er z.B. für alle Methoden des Supertyps Methoden mit entsprechenden Signaturen anbieten. Die Erfüllung dieser syntaktischen Bedingungen reicht zwar aus, damit der Übersetzer das Programm akzeptiert; die Bedingungen bieten aber bestenfalls eine Hilfestellung, um kleinere Fehler zu vermeiden. Sie garantieren nicht, dass sich das Programm so verhält, wie man es erwartet. Beispielsweise könnten wir die Methode `equals`, die zu jeder Klasse gehört (sie wird von `Object` geerbt) und die zwei Objekte auf Gleichheit prüft, in beliebigen Subtypen von `Object` z.B. so implementieren, dass sie die syntaktischen Bedingungen erfüllt, aber immer `true` liefert. Damit würde sich diese Methode und insgesamt der Typ, zu dem sie gehört, nicht erwartungsgemäß verhalten.

Konzeptionell geht Subtyping über die Erfüllung syntaktischer Bedingungen hinaus. Von einem Subtyp-Objekt wird erwartet, dass es sich konform zum Supertyp verhält. Nur dann gilt: Ein Subtyp-Objekt *ist ein* Supertyp-Objekt mit ggf. zusätzlichen Eigenschaften (vgl. Abschn. 3.1). Dass sich Subtyp-Objekte konform zu ihren Supertypen verhalten, liegt heutzutage fast ausschließlich in der Verantwortung der Programmierer. Sie kennen und dokumentieren das gewünschte Verhalten eines Typs und programmieren die Subtypen hoffentlich so, dass diese sich konform verhalten. Um konformes Verhalten maschinengestützt überprüfen zu können, benötigt man eine Spezifi-

kation des Verhaltens von Typen. Die Entwicklung von Spezifikationssprachen und -techniken für diesen Zweck werden z.B. in [LW94] behandelt.

### 3.2.3 Subtyping und Schnittstellen im Kontext parametrischer Typen

Dieser Abschnitt betrachtet zunächst die bisher im nicht parametrischen Kontext behandelten Themen im Kontext parametrischer Schnittstellen und Klassen. Anschließend werden die auf Seite 106 des Kurstextes bereits erwähnten beschränkt parametrische Typen eingeführt. Der letzte Teil geht auf Subtyping-Probleme im Bereich von parametrischen Typen, insbesondere von Behältertypen, ein.

#### 3.2.3.1 Deklaration, Erweiterung und Implementierung parametrischer Schnittstellen

Wie parametrische Klassen können auch Schnittstellen in Java mit Typparametern versehen werden. Die auf Seite 158 eingeführte Syntax zur Deklaration einer Schnittstelle wird daher erweitert zu:

```
[public]opt interface Schnittstellename [ <Liste von Typparametern > ]opt
    [ extends Liste von Schnittstellennamen ggf. mit Typparametern ]opt
{
    Liste von Konstantendeklarationen und Methodensignaturen
}
```

Die Typparameter von Schnittstellen können wie normale Typen bei der Deklaration von Rückgabetypen und Parametertypen verwendet werden. Beispiele für parametrische Schnittstellen findet man z.B. in den Paketen `java.lang` und `java.util`. Einige dieser Schnittstellen stellen wir im Folgenden kurz vor.

```
public interface Comparable<T> {
    // Hier wird T als Parametertyp verwendet
    public int compareTo(T o);
}

public interface Iterator<E> {
    boolean hasNext();
    // Hier wird E als Rueckgabetyp verwendet
    E next();
    void remove();
}

public interface Iterable<T> {
    // Hier wird ein parametrischer Typ
```

```

    // als Rueckgabetyt eingesetzt
    Iterator<T> iterator();
}

// Eine parametrische Schnittstelle erweitert
// eine andere parametrische Schnittstelle
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    boolean add(E o);
    ...
}

// Eine parametrische Schnittstelle erweitert
// eine andere parametrische Schnittstelle
public interface List<E> extends Collection<E> {
    Iterator<E> iterator();
    ...
    boolean add(E o);
    boolean remove(Object o);
    ...
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    ...
    ListIterator<E> listIterator();
    ...
}

```

Die Erweiterung einer parametrischen Klasse durch eine andere parametrische Klasse erfolgt nach dem gleichen Prinzip wie im nicht parametrischen Fall. Dasselbe gilt für parametrische Klassen, die Schnittstellen implementieren. Der folgende Auszug aus der Klassendeklaration der parametrischen Klasse `LinkedList<E>` aus dem Paket `java.util` zeigt, wie eine parametrische Klasse als Erweiterung einer anderen parametrischen Klassen deklariert werden kann und wie eine parametrische Klasse deklariert, dass sie bestimmte parametrische und nicht parametrische Schnittstellen implementiert.

```

public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Queue<E>, Cloneable, java.io.Serializable
{

```

...  
}

Auch bei parametrischen Klassen und Schnittstellen legen die `extends`- und `implements`-Deklarationen explizit eine Typhierarchie fest. Beispielsweise ist für jeden aktuellen Typparameter  $E_0$ , der zur Instanziierung von `LinkedList<E>` verwendet wird, der daraus resultierende Typ `LinkedList<E0 ein direkter Subtyp von AbstractSequentialList<E0 sowie von List<E0, Queue<E0, Cloneable und java.io.Serializable8. Da List<E0 Subtyp von Collection<E0 ist, ist auch LinkedList<E0 Subtyp von Collection<E0 usw. Wäre zur Instanziierung von LinkedList<E> als  $E_0$  z.B. der Typ Integer gewählt worden, so könnte auf Grund der obigen Subtypbeziehung einer Variablen vom Typ List<Integer> eine Variable oder ein Wert vom Typ LinkedList<Integer> zugewiesen werden. Folgender Programmcode wäre also zulässig:`

```
public class SubtypingLinkedList {
    public static void main(String[] args) {
        java.util.List<Integer> li;
        java.util.LinkedList<Integer> lli =
            new java.util.LinkedList<Integer>();
        lli.addLast(new Integer(1));
        lli.addLast(new Integer(2));
        lli.addLast(new Integer(3));

        // Zuweisung einer Variablen vom Typ LinkedList<Integer>
        // an eine Variable vom Typ List<Integer>
        li = lli;

        // Speichert eine weitere Integer-Zahl in lli
        li.add(new Integer(4));

        java.util.Iterator<Integer> it = li.iterator();
        while (it.hasNext()) {
            Integer i = it.next();
            System.out.println(i);
        }
    }
}
```

Da `li` auf `lli` verweist, werden in der `while`-Schleife alle in `lli` gespeicherten `Integer`-Werte, nämlich 1,2,3 und 4, ausgegeben.

---

<sup>8</sup>Diese Schnittstelle werden wir im Zusammenhang mit Objektströmen (Abschnitt 4.3.2.2) noch kennenlernen.

### 3.2.3.2 Beschränkt parametrische Typen

Wie wir bereits in Abschnitt 2.1.6.1 auf Seite 106 erwähnt haben, hat ein parametrischer Typ, so wie wir ihn bisher kennengelernt haben, keinerlei Kenntnisse über seine Typparameter, d.h. er weiß nicht, welche Attribute und Methoden seine Typparameter bereitstellen. Demzufolge kann er z.B. keine Methoden auf Variablen dieser Typen aufrufen. Dies ist in vielen Fällen aber zu einschränkend, insbesondere im Zusammenhang mit Behälterklassen, bei denen die Typparameter den Elementtyp darstellen. Hier wäre es wünschenswert, gemeinsame Eigenschaften aller möglichen Elementtypen deklarieren zu können, sodass die parametrische Klasse vom Vorhandensein dieser Eigenschaften bei den in ihr gespeicherten Elementen ausgehen kann. Als Beispiel rufen wir uns noch einmal die auf Seite 106 bereits angedeutete mögliche Erweiterung der Klasse `LinkedList<ET>` ins Gedächtnis.

Man könnte sich z.B. vorstellen, die Klasse `LinkedList<ET>` von Seite 121 um eine Methode `printAll` zu erweitern, die alle in der Liste gespeicherten Elemente ausdrückt. Um diese Aufgabe erfüllen zu können, müssten die in der Liste gespeicherten Elemente beispielsweise alle über eine Methode `drucken` verfügen, die `LinkedList<ET>` zum Ausdrucken eines jeden Elements verwenden könnte. Würde `LinkedList<ET>` z.B., dass jeder für `ET` eingesetzte Typparameter über eine Methode `drucken` verfügt, wäre dieses Problem gelöst.

*beschränkt  
bzw. gebunden  
parametrischer  
Typ*

Im Zusammenhang mit Subtyping bietet Java 5.0 hierzu eine Lösung an in Form von sogenannten *beschränkt parametrischen Typen*, die auch *gebunden parametrische Typen* genannt werden. Dabei kann in der einfachen Variante für jeden Typparameter eines parametrischen Typs eine „obere Schranke“ in Form eines Supertyps angegeben werden. D.h. alle Typen, die als aktuelle Typparameter eingesetzt werden, müssen Subtyp dieser „oberen Schranke“ sein und damit mindestens die Eigenschaften dieses Supertyps besitzen. In unserem Beispiel der modifizierten Klasse `LinkedList` sollten alle Typen, die für `ET` eingesetzt werden, Subtypen der auf Seite 150 eingeführten Schnittstelle `Druckbar` sein. Dann besäßen diese Typen alle eine Methode `void drucken()`, die innerhalb von `LinkedList` zur Implementierung der neuen Methode `printAll` verwendet werden könnte.

Das folgende Codefragment der Klasse `LinkedList` zeigt, wie der Parametertyp `ET` durch `Druckbar` nach oben beschränkt werden kann und wie die Implementierung von `printAll` aussehen kann.

```
public class LinkedList<ET extends Druckbar> {

    static class Entry<T> {
        ...
    }

    Entry<ET> header = new Entry<ET>(null, null, null);
```

```

...

void printAll() {
    ListIterator it = listIterator();
    while (it.hasNext()) {
        ET elem = it.next();
        elem.drucken();
        System.out.println("=====");
    }
}

class ListIterator {
    ...
}

```

Wie das Beispiel zeigt, wird der Supertyp (die obere Schranke) mit Hilfe des Schlüsselworts `extends` innerhalb der spitzen Klammern deklariert. `extends` wird in jedem Fall verwendet, unabhängig davon, ob es sich beim Supertyp um einen Klassen- oder Schnittstellentyp handelt.

Ein Typparameter kann auch mehrere obere Schranken besitzen. Dies ist z.B. sinnvoll, wenn die für den Typparameter einzusetzenden Typen mehrere Typen erweitern sollen, die selbst in keiner Subtypbeziehung zueinander stehen. Besitzt ein Typparameter mehrere obere Schranken, werden diese durch `&` voneinander getrennt.

Wir betrachten dazu das folgende Beispiel. Die Elementtypen für unsere Klasse `LinkedList` sollen neben der Schnittstelle `Druckbar` auch noch die Schnittstelle `StringConversion` implementieren, die wie folgt deklariert ist:

```

interface StringConversion {
    String convertToString();
}

```

Die in der Schnittstelle deklarierte Methode `convertToString` gibt eine Zeichenkettendarstellung des Elements zurück, auf dem sie aufgerufen wird. Die Klasse `LinkedList` kann dann eine zusätzliche Methode `convertAllElementsToStrings` anbieten, die die Zeichenkettendarstellung aller in ihr gespeicherten Elemente aneinanderreicht und in Form einer einzigen Zeichenkette zurückgibt. Die wesentlichen Codefragmente der so modifizierten Klasse `LinkedList` sehen wie folgt aus:

```

public class LinkedList<ET extends Druckbar & StringConversion> {

    static class Entry<T> {
        ...
    }
}

```



```

Entry<ET> header = new Entry<ET>(null, null, null);
...

void printAll() {
    ...
}

String convertAllElementsToStrings() {
    ListIterator it = listIterator();
    String st = "";
    while (it.hasNext()) {
        ET elem = it.next();
        st = st + " | " + elem.convertToString();
    }
    return st;
}

class ListIterator {
    ...
}

```

### 3.2.3.3 Subtyping bei parametrischen Behältertypen

Im Absatz „Subtyping bei Feldern“ auf Seite 170 haben wir bereits beschrieben, dass sich die Subtypbeziehung zwischen den Elementtypen zweier Felder auf die Felder selbst überträgt, d.h. wenn *CS* ein Subtyp von *CT* ist, dann ist der Feldtyp *CS[]* ein Subtyp des Feldtyps *CT[]*. Daher kann eine Variable oder ein Objekt vom Typ *CS[]* an eine Variable vom Typ *CT[]* zugewiesen werden oder als aktueller Parameter in Methoden verwendet werden, die ein Feld vom Typ *CT[]* verlangen. Das folgende Beispiel demonstriert diese beiden Fälle:

```

public class ArrayTest {

    public static Person[] sort(Person[] personFeld) {
        Person[] personFeldSortiert = new Person[personFeld.length];
        // Speichere die Elemente aus personFeld nach dem
        // Geburtsdatum aufsteigend sortiert nach personFeldSortiert
        ...
        return personFeldSortiert;
    }

    public static void main(String[] args) {
        Angestellte[] angestelltenFeld = new Angestellte[3];
        ...
        Person[] personFeld = angestelltenFeld;
        ...
    }
}

```

```

    // Der sort-Methode wird ein aktueller Parameter
    // vom Typ Angestellte[] uebergeben.
    personFeld = sort(angestelltenFeld);
}
}

```

Die Übertragung der Subtypbeziehung zwischen Elementtypen auf die Subtypbeziehung zwischen den Feldtypen ist sehr hilfreich. So können wir Methoden wie die `sort`-Methode, die ganze Felder als Parameter übernimmt und diese umsortiert, dafür aber nur eine allen Subtypen von `Person` gemeinsame Eigenschaft zum Sortieren nutzt, für Felder vom Typ `Person` schreiben, aber auch für Felder nutzen, deren Elemente von einem Subtyp von `Person` sind, wie z.B. `Angestellte`<sup>9</sup>.

Diese Möglichkeit zur Zuweisung von Feldern mit verschiedenen Elementtypen kann aber auch, wie bereits im Absatz „Subtyping bei Feldern“ auf Seite 170 beschrieben, zu Problemen führen und zwar dann, wenn eine Feldvariable, im Beispiel `personFeld`, ein Feld eines Subtyps referenziert (`angestelltenFeld`) und versucht wird, in diesem Feld ein Objekt eines Supertyps des aktuellen Elementtyps abzuspeichern (s. Zeile (\*)):

```

public class ArrayTest {
    ...

    public static void main(String[] args) {
        Angestellte[] angestelltenFeld = new Angestellte[3];
        ...
        Person[] personFeld = angestelltenFeld;

        // KEIN Compilerfehler; ergibt aber zur Laufzeit
        // eine ArrayStoreException
        (*) personFeld[0] = new Person("Meyer", 19631007);
        ...
    }
}

```

Im Fall von Feldern kann dieses Problem zur Laufzeit erkannt werden, da jedes Feld eine Information über seinen Elementtyp enthält<sup>10</sup>.

<sup>9</sup>In diesem und den folgenden Beispielen greifen wir auf die in Abbildung 3.6 dargestellte Version zurück, in der `Person` und `Angestellte` als Klassen realisiert sind und nicht als Schnittstellen. Die auf Seite 158 für `Person` deklarierten Methoden und die auf Seite 160 für `Angestellte` deklarierten Methoden setzen wir als implementiert voraus.

<sup>10</sup>Dies ist aber im Zusammenhang mit der Löschung von Typparameterinformationen die Ursache für fundamentale Probleme, die im Zusammenhang mit Feldern auftreten, deren Elementtypen aus parametrischen Typen erzeugt wurden. Man kann zwar Feldvariablen mit solchen Elementtypen deklarieren, wie z.B. `ArrayList<Person>[] personFeld`, aber man kann von ihnen keine Instanzen erzeugen (`new ArrayList<Person>[...]` führt zu einem Compilerfehler). Interessierte finden weitere Informationen zu dieser Problematik z.B. in [NW07].

angestelltenFeld weiß also, dass es nur Elemente vom Typ Angestellte (und Subtypen davon) enthalten soll.

Wir könnten jetzt erwarten, dass sich parametrische Behältertypen ähnlich verhalten wie Feldtypen. Wenn also *CS* ein Subtyp von *CT* ist, dann sollte auch ein Behälter *B* mit Elementtyp *CS* ein Subtyp des Behälters *B* mit Elementtyp *CT* sein.

Für den Behältertyp `ArrayList<E>` aus dem Paket `java.util` würde man in Analogie zu den Feldtypen also folgenden Programmcode schreiben:

```
class ArrayListTest {

    public static ArrayList<Person> sort (ArrayList<Person>
                                           personFeld) {
        ArrayList<Person> personFeldSortiert =
                                           new ArrayList<Person>();
        // Speichere die Elemente aus personFeld nach dem
        // Geburtsdatum aufsteigend sortiert
        // nach personFeldSortiert
        return personFeldSortiert;
    }

    public static void main(String[] args) {
        ArrayList<Angestellte> angestelltenFeld =
                                           new ArrayList<Angestellte>();
        ...
        (*) ArrayList<Person> personFeld = angestelltenFeld;
        ...

        (**) personFeld = sort(angestelltenFeld);
    }
}
```

Versucht man, diesen Programmcode zu übersetzen, stellt man fest, dass zunächst in Zeile (\*) ein Compilerfehler auftritt. `ArrayList<Angestellte>` wird im Gegensatz zu Feldtypen **nicht** als Subtyp von `ArrayList<Person>` betrachtet. Aus demselben Grund wird auch in Zeile (\*\*) ein Compilerfehler gemeldet.

Dies mag daran liegen, dass die durch `personFeld` und `angestelltenFeld` referenzierten Listen im Gegensatz zu Feldern zur Laufzeit keine Informationen mehr über ihren Elementtyp enthalten. Auf Grund der Löschung werden `ArrayList<Person>` und `ArrayList<Angestellte>` auf dieselbe Klasse `ArrayList` abgebildet. Daher kann zur Laufzeit nicht mehr festgestellt werden, wenn unzulässigerweise ein `Person`-Objekt in `personFeld` gespeichert werden soll, wie in folgendem Beispiel in Zeile (\*).

```

class ArrayListTest {
    ...
    public static void main(String[] args) {
        ArrayList<Angestellte> angestelltenFeld =
            new ArrayList<Angestellte>();
        ...
        ArrayList<Person> personFeld = angestelltenFeld;

(*) personFeld.add(new Person( "Meyer", 19631007));
        ...
    }
}

```

Möglicherweise wollte man das für Felder verwendete Subtypprinzip aber auch einfach nicht weiterführen.

Dadurch ist uns aber prinzipiell die Möglichkeit genommen worden, eine Methode wie `sort` auch auf Listen anzuwenden, deren Elemente von einem Subtyp von `Person` sind. Da dies eine zu große Einschränkung darstellen würde, wurden in Java 5.0 sogenannte Platzhalter (engl. wildcards) für Typen eingeführt, die in Variablen- und Parameterdeklarationen verwendet werden können. Unsere `sort`-Methode wie auch unsere `main`-Methode könnte mit Hilfe solcher Platzhalter wie folgt geändert werden:

*Typ-Platz-  
halter*

```

class ArrayListTest {

(*) public static ArrayList<Person> sort
    (ArrayList<? extends Person> personFeld) {
    ArrayList<Person> personFeldSortiert = new ArrayList<Person>();
    // Speichere die Elemente aus personFeld nach dem
    // Geburtsdatum aufsteigend sortiert nach
    // personFeldSortiert
    ...
    return personFeldSortiert;
}

    public static void main(String[] args) {
        ArrayList<Angestellte> angestelltenFeld =
            new ArrayList<Angestellte>();
        ...
(**) ArrayList<? extends Person> personFeld = angestelltenFeld;
        ...

(***) personFeld = sort(angestelltenFeld);
    }
}

```

Das Fragezeichen in der Variablendeklaration für `personFeld` in der mit `(**)` markierten Zeile besagt, dass eine Liste mit einem beliebigen Elementtyp, der Subtyp von `Person` ist, an `personFeld` zugewiesen werden kann. Daher ist die Zuweisung `personFeld = angestelltenFeld` jetzt erlaubt.

Analoges gilt für den formalen Parameter der Methode `sort` in der mit (\*) markierten Zeile. Diese Deklaration erlaubt als aktuelle Parameter Listen mit einem beliebigen Elementtyp, der Subtyp von `Person` ist. `angestelltenFeld` kann daher in der mit (\*\*) markierten Zeile jetzt als aktueller Parameter an die `sort`-Methode übergeben werden.

Nicht erlaubt ist dagegen die Anweisung aus Zeile (\*) im Folgebeispiel. Sie erzeugt einen Compilerfehler, da der Compiler nicht weiß, welcher aktuelle Typ für '?' eingesetzt wird. Das kann ein echter Subtyp von `Person` sein, sodass das Speichern von `Person`-Objekten in der Liste möglicherweise nicht erlaubt ist. Analoges gilt für die Folgezeile (\*\*). Für '?' könnte ein echter Subtyp von `Angestellte` eingesetzt werden.

```
class ArrayListTest {
    ...

    public static void main(String[] args) {
        ArrayList<Angestellte> angestelltenFeld =
            new ArrayList<Angestellte>();
        ...
        ArrayList<? extends Person> personFeld = angestelltenFeld;

        // Compilerfehler
        (*) personFeld.add(new Person( "Meyer", 19631007));
        (**) personFeld.add(new Angestellte( "Planck", 18580423,
            18800401, 1));
        ...
    }
}
```

Für weiterführende Anwendungen von Platzhaltern und Einschränkungen bei deren Verwendung sei auf [NW07] und [Krü07] verwiesen.

Dehnt man die Betrachtung der Subtypbeziehung von Behältertypen auf beliebige generische Typen  $G$  aus, so lässt sich festhalten, dass aus  $A \preceq B$  **nicht** folgt, dass dann auch  $G < A > \preceq G < B >$  gilt.

### 3.2.4 Typkonvertierungen und Typtests

Java bietet die Typkonvertierungen nicht nur auf den Basisdatentypen (vgl. Abschn. 1.3.1, S. 33) an, sondern auch zwischen Referenztypen  $S$  und  $T$  mit der Eigenschaft  $S \preceq T$ . Da Objekte des Typs  $S$  an allen Stellen verwendet werden dürfen, an denen Objekte vom Typ  $T$  zulässig sind, sind Typkonvertierungen von  $S$  nach  $T$  nicht notwendig. Von Bedeutung ist die Typkonvertierung von Supertypen auf Subtypen, also von  $T$  nach  $S$ . Ein typisches Beispiel enthält die Klasse `TestIntWrapper` auf S. 172: In der markierten Zeile wird das letzte Element aus der Liste `ll` geholt. Von diesem Element soll das Attribut `value` ausgelesen werden. Listenelemente sind aber vom

Typ `Object` und besitzen im Allgemeinen kein Attribut `value`; der Ausdruck `ll.getLast().value` ist also fehlerhaft. Der Fehler kann – wie oben gezeigt – behoben werden, indem man den Typ des Listenelements nach `Int` konvertiert; `Int` ist ein Subtyp von `Object` und besitzt ein Attribut `value`.

Die Typkonvertierung bewirkt aber nicht nur, dass der Typ des Ausdrucks konvertiert wird, sodass die Typüberprüfungen, die der Übersetzer durchführt, bestanden werden können. Sie führt auch dazu, dass zur Ausführungszeit geprüft wird, ob das Ergebnis der Auswertung des Ausdrucks auch von dem entsprechenden Subtyp ist. Im obigen Beispiel wird also zur Laufzeit getestet, ob das Listenelement vom Typ `Int` ist. Schlägt der Test fehl, wird die Ausführung des Ausdrucks mit einer `ClassCastException` abrupt beendet.

Im Zusammenhang mit einer Typkonvertierung ist es bisweilen sinnvoll, vorher durch einen expliziten Typtest zu prüfen, ob das Ergebnis eines Ausdrucks von einem bestimmten Typ ist. Für derartige Prüfungen stellt Java den *instanceof-Operator* zur Verfügung. Beispielsweise könnte man die markierte Zeile aus der Klasse `TestIntWrapper` durch folgendes fast äquivalentes Programmfragment ersetzen:

```
int i;
Object ov = ll.getLast();
if (ov instanceof Int)
    i = ((Int)ov).value;
else
    throw new ClassCastException();
```

Zunächst wird der Aufruf `ll.getLast()` ausgewertet. Falls das Ergebnis vom Typ `Int` ist, wird `ov` zum Typ `Int` konvertiert und das Attribut `value` ausgelesen; wegen des vorgeschalteten Typtests kann die Typkonvertierung nicht fehlschlagen. Andernfalls wird die Ausführung mit einer `ClassCastException` abrupt beendet. Der kleine semantische Unterschied zwischen der markierten Zeile aus der Klasse `TestIntWrapper` und dem Programmfragment rührt daher, dass der `instanceof-Operator` `false` liefert, wenn das erste Argument die Referenz `null` ist; andererseits gehört `null` zu jedem Referenztyp.

Auch bei den expliziten Typtests spielen parametrische Typen eine Sonderrolle. Java 5.0 erlaubt es **nicht**, den Operator `instanceof` auf parametrische Typen oder Typen, die aus parametrischen Typen generiert wurden, anzuwenden. Dazu betrachten wir folgendes Beispiel.

```
public class CastAndTypeTest {
    public static void main(String[] args) {
        Object[] ao = new Object[2];
        ao[0] = new ArrayList<Integer>();
```

```

    ao[1] = new ArrayList<String>();

    for (int i = 0; i < ao.length; i++) {
        Object o = ao[i];

        // Compilerfehler: Ungültiger generischer
        // Typ fuer instanceof ...
    (*)    if (o instanceof ArrayList<Integer>)
    (**)        ((ArrayList<Integer>)o).add(new Integer(1));

        // Compilerfehler: Ungültiger generischer
        // Typ fuer instanceof ...
    (***)    if (o instanceof ArrayList<String>)
    (****)        ((ArrayList<String>)o).add("Erster Eintrag");

    }

    System.out.println(ao[0]);
    System.out.println(ao[1]);
}

```

Um sicherzustellen, dass nur Integer-Objekte in Objekten vom Typ `ArrayList<Integer>` gespeichert werden und nur String-Objekte in Objekten vom Typ `ArrayList<String>`, würden wir versuchen, dies durch Anwendung des `instanceof`-Operators in den Zeilen (\*) und (\*\*) zu garantieren. Dieser Typtest wird vom Compiler aber nicht zugelassen. Aufgrund der Löschung kann zur Laufzeit nämlich nicht mehr festgestellt werden, ob z.B. `ao[0]` ein `ArrayList`-Objekt mit Elementen vom Typ `Integer`, `String` oder sonst eines Typs referenziert.

Lassen wir die Typtests weg, so meldet der Übersetzer lediglich je eine Warnung<sup>11</sup> in den Zeilen (\*\*) und (\*\*\*\*). Er führt die gewünschte Typkonvertierung aber in jedem Fall durch. Zur Laufzeit kann dann zwar noch geprüft werden, ob `ao[0]` und `ao[1]` je ein Objekt des Typs `ArrayList` referenzieren. (Wenn nicht, erfolgt eine `ClassCastException`.) Es kann aber nicht geprüft werden, welche Elementtypen für diese Listen vorgesehen waren. Jede der beiden `add`-Anweisung wird daher unabhängig vom Elementtyp auf `o` durchgeführt. Beide Listen enthalten zum Schluss daher dieselben Elemente, nämlich ein `Integer`-Objekt mit dem Wert 1 und ein `String`-Objekt mit dem Wert „Erster Eintrag“.

<sup>11</sup>`CastAndTypeTest.java` uses unchecked or unsafe operations.  
Note: Recompile with `Xlint:unchecked` for details.

### 3.2.5 Unterschiedliche Arten von Polymorphie

Polymorphie bedeutet Vielgestaltigkeit. Im Zusammenhang mit Programmiersprachen spricht man von *Polymorphie*, wenn Programmkonstrukte oder Programmteile für Objekte (bzw. Werte) mehrerer Typen einsetzbar sind. In diesem Unterabschnitt sollen die im Zusammenhang mit Java behandelten Arten von Polymorphie kurz verglichen werden.

*Polymorphie*

**Subtyp-Polymorphie.** In einer Liste mit Elementtyp `Object` können Objekte beliebigen Typs eingefügt werden. (Eine derartige Listenklasse erhält man beispielsweise, wenn man in der Klasse `LinkedList` von Abb. 2.9, S. 101, den Typ `ListElem` durch `Object` ersetzt. Auf die parametrische Variante gehen wir erst im Absatz über parametrische Polymorphie ein.) Insbesondere kann eine solche Liste Elemente ganz unterschiedlichen Typs enthalten. Man spricht deshalb auch von einer *inhomogenen* Liste. Listen bzw. Behälter sind nur Beispiele dafür, wie Subtyping genutzt werden kann, um eine Implementierung so zu formulieren, dass sie für Objekte unterschiedlicher Klassen funktioniert.

Die Form von Polymorphie, die man durch Subtyping erreicht, nennt man *Subtyp-Polymorphie*. Sie bietet ein hohes Maß an Flexibilität. Allerdings ist es oft schwierig, die Flexibilität im Nachhinein einzuschränken. Um dies zu illustrieren, nehmen wir beispielsweise an, dass wir Listen von `String`-Objekten benötigen. Selbstverständlich können wir dafür eine Listenklasse mit Elementtyp `Object` verwenden wie die oben erwähnte Klasse `LinkedList`. Allerdings müssen wir dann jedes `String`-Objekt, das wir uns aus der Liste holen, vor seiner weiteren Verwendung zum Typ `String`<sup>12</sup> konvertieren, wie die drittletzte und letzte Zeile in folgendem Fragment zeigen (vgl. Klasse `Test` auf S. 105):

*Subtyp-Polymorphie*

```
LinkedList ls = new LinkedList();
ls.addLast("letztes Element");
((String) ls.getLast()).indexOf("Elem"); // liefert 8
ls.addLast( new Real() );                // kein Uebersetzungsfehler
((String) ls.getLast()).indexOf("Elem"); // Laufzeitfehler
```

Wenn wir wissen, dass die Liste `ls` nur für `Strings` verwendet werden soll, wäre es günstig, einen Listentyp zu besitzen, der nur `Strings` verarbeitet, wie im Fall der parametrischen Variante der Klasse `LinkedList` (s.u.). Dann können zum einen die Typkonvertierungen und die Gefahr von Laufzeitfehlern vermieden werden. Zum anderen kann der Übersetzer die Stellen ent-

<sup>12</sup>Die Methode `public int indexOf(String str)` der Klasse `String` prüft, ob der ihr als Zeichenkette übergebene Parameter `str` in der aktuellen Zeichenkette als Teil vorkommt. Wenn ja, liefert sie den Index des ersten Buchstabens des ersten Vorkommens von `str` in der aktuellen Zeichenkette zurück.



decken, an denen wir aus Versehen Objekte, die nicht vom Typ `String` sind, in die Liste eingetragen haben (vgl. vorletzte Zeile im obigen Fragment).

*parametrische  
Polymorphie*

**Parametrische Polymorphie.** Polymorphie lässt sich auch dadurch realisieren, dass man Programmkonstrukte – in objektorientiertem Kontext sind das dann meist Klassen und Methoden – mit Typen parametrisiert. Diese sogenannte *parametrische Polymorphie* wird von den meisten modernen Programmiersprachen unterstützt. In Abschn. 2.1.6 haben wir diese Technik bereits erläutert. Parametrische Polymorphie bietet weniger Flexibilität als Subtyp-Polymorphie. Beispielsweise lassen sich mit Sprachen, die **nur** parametrische Polymorphie unterstützen, keine inhomogenen Behälter realisieren. Andererseits gestattet parametrische Polymorphie bei vielen Anwendungen eine leistungsfähigere Typanalyse zur Übersetzungszeit. Zur direkten Gegenüberstellung der beiden Arten von Polymorphie betrachten wir eine Realisierung des obigen Programmfragments mittels der parametrischen Listenklasse aus Abschn. 2.1.6:

```
LinkedList<String> ls = new LinkedList<String>();
ls.addLast("letztes Element");
ls.getLast().indexOf("Elem"); // liefert 8
ls.addLast( new Real() );      // Uebersetzungsfehler !!
ls.getLast().indexOf("Elem");
```

Oft werden beide Ansätze kombiniert, sodass auch Objekte eines Subtyps eines Typparameters `T` an allen Stellen erlaubt sind, an denen Objekte des Typs `T` erlaubt sind. So lassen sich in einer Liste `LinkedList<Person>` z.B. auch `Angestellte`-Objekte speichern.

Ein ausführlicher Vergleich von Subtyp- und parametrischer Polymorphie findet sich in [Mey00], Kap. 19.

*beschränkt  
parametrische  
Polymorphie*

**Beschränkt parametrische Polymorphie.** Beschränkt parametrische Polymorphie verbindet Subtyp-Polymorphie mit parametrischer Polymorphie. Sie ermöglicht es, Typparameter von parametrischen Typen durch Angabe von Supertypen zu beschränken. Für einen durch einen Typ `S` beschränkten Typparameter dürfen nur solche Typen `T` eingesetzt werden, die Subtypen von `S` sind. Die Verwendung solcher Typen bringt dem Anwender zunächst keinen Vorteil vor parametrischen Typen wie oben beschrieben. Vorteile bringt dieser Ansatz vor allem bei der Implementierung solcher beschränkt parametrischer Typen. In der Implementierung kann nämlich das Wissen ausgenutzt werden, dass Objekte eines Typs `T`, der für einen Typparameter `ET` eingesetzt wird, der selbst nach oben durch einen Supertyp `S` beschränkt ist, mindestens über alle Eigenschaften von `S` verfügen. Diese Eigenschaften können dann in der Implementierung ausgenutzt werden.

Als Beispiel hatten wir die Klasse `LinkedList` kennengelernt, deren Elementtyp `ET` nach oben durch den Schnittstellentyp `Druckbar` beschränkt wurde. Dadurch wusste `LinkedList`, dass alle ihre Elemente eine Methode `drucken` zur Verfügung stellen und konnte mit Hilfe dieses Wissens die Methode `printAll` implementieren. Letztendlich profitiert dann auch ein Nutzer dieses Typs von der Beschränkung, da er die Methode `printAll` schon zur Verfügung gestellt bekommt und nicht selbst implementieren muss.

**Ad-hoc-Polymorphie.** Das Überladen von Operatoren bzw. Methodennamen bezeichnet man oft als *Ad-hoc-Polymorphie*. Überladung erlaubt es, Operationen, die man konzeptionell identifiziert, für ggf. verschiedene Typen unter dem gleichen Namen anzubieten – z.B. eine `plus`-Operation. Gerade in Kombination mit Subtyping wird das konzeptionelle Identifizieren verschiedener Methoden deutlich. Man betrachte die folgenden beiden `plus`-Methoden für komplexe und reelle Zahlen, wobei die Klasse `Real` Subtyp der Klasse `Complex` ist. Die Klasse `Real` bietet neben der von `Complex` geerbten `plus`-Methode eine weitere `plus`-Methode an, die nur reellwertige Parameter zulässt.

*Ad-hoc-  
Polymorphie*

```
Complex plus( Complex c ) { ... }
Real      plus( Real r    ) { ... }
```

Beide Methoden müssen Argumente vom Typ `Real` bearbeiten können. Bei gleichem Argument sollten sie auch das gleiche Ergebnis liefern.

Im Gegensatz zu den anderen Formen von Polymorphie führt Überladung aber nur zu einer konzeptionellen Identifikation und nicht dazu, dass derselbe Programmcode für die Bearbeitung unterschiedlicher Typen herangezogen wird. Überladung wird zur Übersetzungszeit aufgelöst, also nicht dynamisch gebunden (siehe auch Abschn. 3.3.5). Technisch gesehen bietet sie dementsprechend nur größere Flexibilität bei der Namensgebung. Deshalb sagt man auch, dass Überladung keine Form *echter* Polymorphie ist.

### 3.2.6 Programmieren mit Schnittstellen

Eine Schnittstelle deklariert einen abstrakten Typ von Objekten und die Signaturen der zugehörigen Methoden. Schnittstellentypen lassen sich für unterschiedliche programmtechnische Aufgabenstellungen verwenden, je nachdem, ob man stärker die Objekte oder die Methoden des Typs in den Vordergrund stellt. Bisher haben wir hauptsächlich die klassische Anwendung von Schnittstellentypen betrachtet, bei der die Objekte im Mittelpunkt des Interesses stehen.

Dieser Abschnitt beschreibt zunächst drei weitere solcher klassischen Schnittstellen: die Schnittstellen `Iterable<T>` und `Comparable<T>` aus dem Paket `java.lang` und die Schnittstellen `Iterator<E>` aus dem Paket

`java.util`. Diese Schnittstellen sind besonders interessant im Zusammenhang mit Behältertypen.

Anschließend wird darauf eingegangen, wie man Schnittstellen zur Realisierung von Aufzählungstypen einsetzen kann und welche anderen Möglichkeiten es ab der Version 5.0 gibt, Aufzählungstypen zu deklarieren.

Im Anschluss daran wird an zwei Beispielen erläutert, wie man Schnittstellentypen auch anders verwenden kann, nämlich um Methoden als Parameter zu übergeben und „callback“-Mechanismen zu realisieren. Dabei werden wir auch kurz auf die Deklaration von lokalen und anonymen Klassen eingehen.

### 3.2.6.1 Die Schnittstellen `Iterable`, `Iterator` und `Comparable`

Dieser Abschnitt beschreibt drei Schnittstellen, die sehr häufig im Kontext von Behältertypen verwendet werden. Die beiden ersten Schnittstellen erleichtern das Iterieren über Behälter und erlauben es, die in Abschnitt 1.3.1.3 auf Seite 37 eingeführte neue Variante der `for`-Schleife (`for-each`) dafür zu verwenden. Hier ist zunächst die Deklaration der Schnittstelle `Iterator` aus dem Paket `java.util`.

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

Zusätzlich zu den Methoden `hasNext()` und `next()`, die wir im Kontext von `LinkedList` bereits kennengelernt haben, bietet diese Schnittstelle noch eine Methode `remove` an, die bei der Implementierung auch einen leeren Rumpf erhalten kann. Wird diese Methode vom implementierenden Behälter aber mit Leben gefüllt, soll sie das Element löschen, das als Letztes vom `Iterator` über `next()` zurückgegeben wurde.

Die Schnittstelle `Iterable` besitzt eine einzige Methode, die einen `Iterator` für einen Behälter mit Elementtyp `T` zurückgibt.

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

Implementiert eine Klasse direkt oder indirekt diese Schnittstelle, kann zum Iterieren über die in Objekten dieser Klasse gespeicherten Elemente die `for-each` Schleife verwendet werden. Zur Demonstration ändern wir die Klasse `LinkedList<ET>` von Seite 121 so ab, dass sie die Schnittstelle `java.lang.Iterable` implementiert und ihre innere Klasse `ListIterator` die Schnittstelle `java.util.Iterator`. Die Klasse

LinkedListTest zeigt, wie die in Listen vom Typ `LinkedList<...>` gespeicherten Elemente jetzt mit Hilfe der for-each Schleife ausgegeben werden können.

```
class LinkedList<ET> implements Iterable<ET> {

    static class Entry<T> {
        ...
    }

    Entry<ET> header = new Entry<ET>(null, null, null);
    int size = 0;
    ...

    class ListIterator implements Iterator<ET> {
        private int nextIndex = 0;
        private Entry<ET> next = header.next;

        public boolean hasNext() {
            return nextIndex != size;
        }
        public ET next() {
            if (nextIndex == size)
                throw new NoSuchElementException();
            ET elem = next.element;
            next = next.next;
            nextIndex++;
            return elem;
        }
        public void remove() {}
    }

    public Iterator<ET> iterator() {
        return new ListIterator();
    }
}

public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList<String> ls = new LinkedList<String>();
        ls.addLast("erstes Element");
        ls.addLast("letztes Element");
        for (String st : ls) System.out.println(st);

        LinkedList<Integer> li = new LinkedList<Integer>();
        li.addLast(new Integer(1));
        li.addLast(new Integer(2));
        li.addLast(new Integer(3));
        for (Integer i : li) System.out.println(i);
    }
}
```

Die Schnittstelle `Comparable<T>` aus dem Paket `java.lang` bietet eine einzige Methode `compareTo` an. Diese Methode dient dazu, das ihr als Parameter übergebene Objekt `o` mit dem Objekt zu vergleichen, auf dem die Methode aufgerufen wird. Sie soll eine negative ganze Zahl zurückgeben, wenn das `this`-Objekt „kleiner“ ist als `o`, Null, wenn das `this`-Objekt „gleich“ `o` ist und eine positive ganze Zahl, wenn das `this`-Objekt „größer“ ist als `o`. Dabei hängt es vom Typ der Objekte ab, was „größer“, „kleiner“ und „gleich“ bedeutet.

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Diese Schnittstelle wird sehr häufig verwendet in Fällen, in denen man ein Sortierkriterium benötigt wie z.B. im Fall von Binärbäumen, sortierten Listen oder sortierten Feldern. Methoden, die beim Einfügen, Löschen oder expliziten Sortieren das Sortierkriterium der Elemente beachten müssen, können dann unabhängig vom konkreten Elementtyp formuliert werden, sofern dieser nur die Schnittstelle `Comparable` implementiert.

### 3.2.6.2 Schnittstellen und Aufzählungstypen

Wie bereits auf Seite 159 erwähnt, wurden benannte Konstanten bis zur Java-Version 1.4 häufig – wie im folgenden Beispiel – zur Realisierung von Aufzählungstypen verwendet:

```
interface Farben {
    byte rot    = 0;
    byte gruen  = 1;
    byte blau   = 2;
    byte gelb   = 3;
}
```

Ab der Version 5.0 bietet Java nun eigene Aufzählungstypen an, sodass der Umweg über Schnittstellentypen nicht mehr benötigt wird. Der oben simulierte Aufzählungstyp kann dann wie folgt mit Hilfe des neuen Schlüsselworts `enum` direkt deklariert werden:

```
enum Farben {rot, gruen, blau, gelb};
```

Diese Deklaration bewirkt, dass ein Datentyp `Farben` generiert wird, dessen Objekte die Werte `Farben.rot`, `Farben.gruen`, `Farben.blau` und `Farben.gelb` annehmen können und keine anderen.

Dies bietet gegenüber der Schnittstellenvariante u.a. mehr Typsicherheit. Während im Fall der Realisierung von Aufzählungstypen mit Schnittstellen

Methoden, die Werte von Aufzählungstypen als Parameter entgegennehmen, nur vom Typ der Konstanten (in unserem Beispiel `byte`) für ihre Parametertypen ausgehen können, können solche Methoden bei Verwendung von Aufzählungstypen diesen Aufzählungstyp als ihren Parametertyp wählen. So kann der Compiler prüfen, ob als aktuelle Parameter nur die zugelassenen Werte übergeben werden. Im Fall von Schnittstellen kann dies nicht mehr entschieden werden. In unserem Beispiel könnten einer Methode

```
void farbVergleich( byte farbel, byte farbe2);,
```

die zwei übergebene Farben auf Gleichheit prüft, Werte außerhalb des gültigen Bereichs von 0 bis 3 als Parameter übergeben werden. Greift die Methode auf den Aufzählungstyp für ihre Parametertypen zurück wie in

```
void farbVergleich( Farben farbel, Farben farbe2);,
```

können ihr als aktuelle Parameter nur die Werte `Farben.rot`, `Farben.gruen`, `Farben.blau` und `Farben.gelb` übergeben werden. Analoges gilt für Variablen, die Werte von Aufzählungstypen aufnehmen sollen.

Jeder Aufzählungstyp wird als nicht erweiterbare Klasse realisiert und seine Werte als Objekte dieses Typs. Aufzählungstypen bzw. deren Objekte besitzen noch viele weitere nützliche Eigenschaften, die im Folgenden aufgelistet werden:

- Sie besitzen eine Methode `toString`, die den Namen des Wertes, für den sie aufgerufen wird, als Zeichenkette zurückgibt.
- Sie besitzen eine Methode `ordinal`, die für den Wert, für den sie aufgerufen wird, eine den Wert identifizierende `int`-Zahl zurückgibt.
- Zwei Objekte eines Aufzählungstyps können mittels `equals` auf Gleichheit geprüft werden.
- Aufzählungstypen implementieren die Schnittstellen `Comparable` und `Serializable`<sup>13</sup>.
- Aufzählungstypen können in `switch`-Anweisungen verwendet werden.
- Jeder Aufzählungstyp besitzt eine Methode `values`, die seine möglichen Werte in Form eines Feldes zurückgibt, über das man dann mit Hilfe der `for-each` Schleife iterieren kann.

Das folgende, etwas modifizierte Beispiel aus [Krü07] demonstriert, dass für Objekte von Aufzählungstypen die Methode `equals` zur Verfügung steht (s. `farbVergleich`) und dass Aufzählungstypen in `switch`-Anweisungen verwendet werden können (s. `toRGB`).

<sup>13</sup>Diese Schnittstelle werden wir im Zusammenhang mit Objektströmen (Abschnitt 4.3.2.2) noch kennenlernen.

```

class RGBFarben {
    enum Farben {rot, gruen, blau, gelb};

    public static void farbVergleich(Farben f1, Farben f2)
    {
        System.out.print(f1);
        System.out.print(f1.equals(f2) ? " = " : " != ");
        System.out.println(f2);
    }

    public static String toRGB(Farben f)
    {
        String ret = "?";
        switch (f) {
            case rot:    ret = "(255,0,0)"; break;
            case gruen:  ret = "(0,255,0)"; break;
            case blau:   ret = "(0,0,255)"; break;
            case gelb:   ret = "(255,255,0)"; break;
        }
        return ret;
    }
}

```

Weitere Eigenschaften von Aufzählungstypen, u.a. das Iterieren über deren Werte, demonstriert die folgende Klasse Aufzaehlungen:

```

public class Aufzaehlungen {
    public static void main(String[] args)
    {
        // Variablen vom Aufzaehlungstyp deklarieren
        // und verwenden
        RGBFarben.Farben f1 = RGBFarben.Farben.rot;
        RGBFarben.Farben f2 = RGBFarben.Farben.blau;
        RGBFarben.Farben f3 = RGBFarben.Farben.rot;

        // toString() liefert einen Aufzaehlungswert
        // in Form einer Zeichenkette
        System.out.println("--");
        System.out.println(f1);
        System.out.println(f2);
        System.out.println(f3);

        // farbVergleich ruft intern equals auf
        System.out.println("--");
        RGBFarben.farbVergleich(f1, f2);
        RGBFarben.farbVergleich(f1, f3);
        RGBFarben.farbVergleich(f2, f3);
    }
}

```

```

    RGBFarben.farbVergleich(f1, f1);

    // Die Methode values() liefert ein Feld mit
    // allen Farbwerten zurueck, ueber das mit Hilfe
    // der for-each Schleife iteriert werden kann.
    System.out.println("--");
    for (RGBFarben.Farben f : RGBFarben.Farben.values()) {
        System.out.println(f + "=" + RGBFarben.toRGB(f));
    }
}
}

```

Das oben stehende Programm erzeugt folgende Ausgabe:

```

--
rot
blau
rot
--
rot != blau
rot = rot
blau != rot
rot = rot
--
rot=(255,0,0)
gruen=(0,255,0)
blau=(0,0,255)
gelb=(255,255,0)

```

Im Gegensatz zu anderen Sprachen mit Aufzählungstypen erlaubt es Java, dass man Aufzählungstypen bei der Deklaration auch mit weiteren Attributen und Methoden versieht, um zusätzliche Funktionalität anzubieten. Das folgende Beispiel (s. auch [Krü07]) zeigt eine solche Deklaration. Der Aufzählungstyp erhält drei zusätzliche Attribute, die die RGB-Farben zu jedem Wert des Aufzählungstyps speichert. Zur Initialisierung dieser Werte enthält die Deklaration auch einen Konstruktor. Die Wertdeklarationen für die Farben, z.B. `rot`, müssen dann aktuelle Parameter für diesen Konstruktoraufbau enthalten.

```

enum Farben2 {

    // Da Farben2 einen Konstruktor hat, muessen bei
    // der Deklaration der Werte, die ja Objekte des
    // Aufzaehlungstyps sind, auch aktuelle
    // Konstruktor-Parameter angegeben werden.
    rot(255, 0, 0),

```



```

    gruen(0, 255, 0),
    blau(0, 0, 255),
    gelb(255, 255, 0),
    weiss(255, 255, 255);

    private final int r;
    private final int g;
    private final int b;

    Farben2(int r, int g, int b)
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public String toRGB()
    {
        return "(" + r + ", " + g + ", " + b + ")";
    }
}

```

Die Werte dieses Typs kennen jetzt auch ihren RGB-Wert, den sie bei Bedarf über die Methode `toRGB` als Zeichenkette liefern können. Für weitere Details bzgl. Aufzählungstypen siehe z.B. [Krü07] und [NW07].

### 3.2.6.3 Schnittstellen zur Realisierung von Methodenparametern

Java kennt im Gegensatz zu z.B. C# keine Methodentypen. In Java gibt es daher keine Möglichkeit, Methoden direkt als Parameter an andere Methoden zu übergeben. Man kann eine solche Parameterübergabe aber mit Hilfe von Schnittstellen simulieren. Im Allgemeinen kann man dazu wie folgt vorgehen:

1. Definiere einen Schnittstellentyp mit einer Methode, deren Signatur den Anforderungen an die Methoden entspricht, die als Parameter übergeben werden sollen. (s. Schnittstelle `FilterPraedikat`)
2. An den Stellen, an denen der formale Methodenparameter deklariert werden muss, benutze den Schnittstellentyp als formalen Parametertyp. (s. Methode `filter` in Klasse `FilterMain` aus Abb. 3.7)
3. An den Stellen, an denen der Methodenparameter aufgerufen werden soll, rufe die Schnittstellenmethode auf dem formalen Parameter auf. (s. Anweisung `fp.test(strs[i])` in der Methode `filter` von Klasse `FilterMain` aus Abb. 3.7)

4. Realisiere Methoden, die als aktuelle Parameter übergeben werden sollen, im Rahmen von Klassen, die den Schnittstellentyp implementieren. (s. Klassen `PrefixAoderB` und `Lexik` aus Abb. 3.7)

Als Beispiel betrachten wir eine Methode `filter`, die aus einem Feld von Zeichenreihen diejenigen Zeichenreihen auswählt, die einen Test bestehen. Der durchzuführende Test in Form einer Methode soll der `filter`-Methode eigentlich als Parameter übergeben werden. Da dies in Java nicht direkt möglich ist, gehen wir wie folgt vor.

Zunächst definieren wir einen Schnittstellentyp für die Testmethode:

```
interface FilterPraedikat {  
    boolean test( String s );  
}
```

Darauf aufbauend können wir eine parametrisierte Methode `filter` realisieren, die ein `FilterPraedikat`-Objekt als Parameter nimmt und darüber die zugehörige Testmethode aufruft (siehe Abb. 3.7). Wir demonstrieren die Anwendung der Methode `filter` anhand zweier Testmethoden: Die Methode `test` der Klasse `PrefixAoderB` wählt diejenigen Zeichenreihen aus, die mit „A“ oder „B“ beginnen. Die Verwendung der Testmethode aus der Klasse `Lexik` ergibt eine lexikographisch geordnete Sequenz von Zeichenreihen, die aus der übergebenen Sequenz entsteht, wenn man alle Zeichenreihen streicht, die lexikographisch vor ihrem Vorgänger anzuordnen wären. Darüber hinaus zeigt Abbildung 3.7 eine Klasse `FilterMain`, die die Anwendung der Methode `filter` mittels der beiden Testmethoden illustriert.

Vergleicht man die obige Technik mit der Verwendung von Prozedurparametern bzw. von Zeigern auf Prozeduren wie man sie aus Pascal, C oder C# kennt, so ergeben sich im Wesentlichen zwei Unterschiede:

1. Einerseits ist der syntaktische Aufwand bei der demonstrierten Technik größer: Der Schnittstellentyp muss extra deklariert werden und die anzuwendenden Methoden müssen innerhalb einer Klasse deklariert werden, die diesen Typ implementieren.
2. Andererseits ist die demonstrierte Technik flexibler als Prozedur- bzw. Methodenzeiger, da sie es ermöglicht, für die Methoden einen (lokalen) Ausführungskontext zu definieren. Beispielsweise enthält die Klasse `Lexik` das Attribut `aktMax`, in dem das aktuelle Maximum der zurückliegenden Aufrufe von `test` gespeichert werden kann.

```

class PrefixAoderB implements FilterPraedikat {
    public boolean test( String s ) {
        return s.length() > 0    &&
            ( s.charAt(0)=='A' || s.charAt(0)=='B' ) ;
    }
}

class Lexik implements FilterPraedikat {
    String aktMax = "";
    public boolean test( String s ) {
        if( s.compareTo( aktMax ) >= 0 ) {
            aktMax = s;
            return true;
        } else return false;
    }
}

public class FilterMain {
    static void printStringArray( String[] strs ) {
        for( int i = 0; i< strs.length; i++ )
            System.out.println( strs[i] );
    }

    static String[] filter( String[] strs, FilterPraedikat fp ){
        int i, j = 0;
        String[] aux = new String[ strs.length ];
        for( i = 0; i< strs.length; i++ ) {
            if( fp.test(strs[i]) ) { // <--- hier ist der Aufruf
                                    //          der Testmethode
                aux[j] = strs[i];
                j++;
            }
        }
        String[] ergebnis = new String[ j ];
        System.arraycopy( aux, 0, ergebnis, 0, j );
        return ergebnis;
    }

    public static void main(String[] args) {
        System.out.println("\nAusgabe von PrefixAoderB:");
        printStringArray( filter( args, new PrefixAoderB() ) );

        System.out.println("\nAusgabe von Lexik:");
        printStringArray( filter( args, new Lexik() ) );
    }
}

```

Abbildung 3.7: Schnittstellen zur Realisierung von Methoden als Parameter

### 3.2.6.4 Beobachter und lokale Klassen

Ein häufig verwendetes Entwurfsmuster<sup>14</sup> in der objektorientierten Programmierung ist das sogenannte *Beobachtermuster*. Dabei können sich Beobachter-Objekte bei einem Objekt anmelden. Tritt an diesem Objekt ein Ereignis auf, benachrichtigt es alle bei ihm angemeldeten Beobachter. In Kap. 5 werden wir sehen, wie dieses Muster bei der Realisierung von Bedienoberflächen eingesetzt wird. Hier konzentrieren wir uns auf die programmtechnischen Aspekte, nämlich die Anwendung von Schnittstellentypen zur abstrakten Beschreibung von Beobachtern.

*Beobachter-  
muster*

Als Beispiel betrachten wir Beobachter, die die Kurse von Aktien beobachten wollen. Immer wenn der Kurs einer Aktie steigt oder fällt, sollen alle bei der Aktie gemeldeten Beobachter benachrichtigt werden. Dafür stellen die Beobachter die Methoden `steigen` und `fallen` zur Verfügung, mit denen die Aktie ihre Beobachter über die entsprechenden Ereignisse benachrichtigt. Bei der Benachrichtigung wird auch eine Referenz auf die Aktie übergeben, die die Veränderung meldet:

```
interface Beobachter {  
    void steigen( Aktie a );  
    void fallen( Aktie a );  
}
```

Eine Aktie hat einen Namen, einen Kurswert und eine Liste der bei ihr gemeldeten Beobachter. Zur Implementierung der Liste verwenden wir die Klasse `ArrayList` aus dem Java-Bibliothekspaket `java.util`; zum Durchlaufen der Liste benutzen wir die `for-each` Schleife. Die Methode zum Setzen des Kurswerts übernimmt auch die Benachrichtigung: Ist der Kurswert gestiegen, wird für jeden gemeldeten Beobachter die Methode `steigen` aufgerufen; ist er gesunken, die Methode `fallen` (siehe Abb. 3.8).

Wichtig ist in diesem Zusammenhang, dass der Schnittstellentyp `Beobachter` eine saubere Trennung zwischen der Benachrichtigung der Beobachter durch die Aktie und der Reaktion der Beobachter auf die Benachrichtigung bewirkt. Einerseits sind die Beobachter der Verpflichtung enthoben, immer wieder nach dem Kursstand der Aktien zu schauen. Andererseits brauchen die Aktien sich nicht um die möglicherweise sehr verschiedenen Beobachter zu kümmern.

---

<sup>14</sup>*Entwurfsmuster* bieten Lösungen für wiederkehrende Problemstellungen beim objektorientierten Entwurf; siehe [GHJV95].

```
import java.util.*;

public class Aktie {
    private String name;
    private int kursWert;
    private ArrayList<Beobachter> beobachterListe;

    Aktie( String n, int anfangsWert ){
        name = n;
        kursWert = anfangsWert;
        beobachterListe = new ArrayList<Beobachter>();
    }

    public void anmeldenBeobachter( Beobachter b ) {
        beobachterListe.add( b );
    }
    public String getName(){
        return name;
    }
    public int getKursWert(){
        return kursWert;
    }
    void setKursWert( int neuerWert ){
        int alterWert = kursWert;
        kursWert = neuerWert>0 ? neuerWert : 1 ;

        if( kursWert > alterWert ) {
            for (Beobachter b : beobachterListe)
                b.steigen( this );
        }
        else {
            if (kursWert < alterWert) {
                for (Beobachter b : beobachterListe) {
                    b.fallen(this);
                }
            }
        }
    }
}
```

Abbildung 3.8: Aktien benachrichtigen Objekte

Als Anwendungsbeispiel betrachten wir hier zwei recht unterschiedliche Beobachter. Ihr Verhalten wird von den Klassen `Boersianer1` und `Boersianer2` beschrieben. Der erste Börsianer kauft von einer beobachteten Aktie, wenn deren Kurs unter 300 fällt und er noch keine besitzt, und verkauft, wenn der Kurs über 400 steigt:

```
class Boersianer1 implements Beobachter {
    private boolean besitzt = false;

    public void fallen( Aktie a ) {
        if( a.getKursWert() < 300 && !besitzt ) {
            System.out.println("Kauf von "+a.getName() );
            besitzt = true;
        }
    }

    public void steigen( Aktie a ) {
        if( a.getKursWert() > 400 && besitzt ) {
            System.out.println("Verkauf von "+a.getName());
            besitzt = false;
        }
    }
}
```

Der zweite Börsianer ermittelt nur den maximalen Kurswert der beobachteten Aktien:

```
class Boersianer2 implements Beobachter {
    private int maximum = 0;

    public void steigen( Aktie a ) {
        if( a.getKursWert() > maximum ) {
            maximum = a.getKursWert();
            System.out.println("Neues Maximum "+a.getName()
                               + ": " + maximum );
        }
    }

    public void fallen( Aktie a ) { }
}
```

Das Zusammenspiel von Aktien und Beobachtern demonstriert das folgende Rahmenprogramm. Es definiert zwei Aktien und die Börsianer `peter` und `georg`. In einer nicht terminierenden Schleife wird dann ein zufälliger Kursverlauf der Aktien erzeugt:

```

public class Main1 {
    public static void main(String argv[]){
        Aktie vw    = new Aktie( "VW",    354 );
        Aktie bmw   = new Aktie( "BMW",   548 );

        Beobachter peter = new Boersianer1();
        vw .anmeldenBeobachter( peter );

        Beobachter georg = new Boersianer2();
        vw .anmeldenBeobachter( georg );
        bmw.anmeldenBeobachter( georg );

        while( true ){
            System.out.print("VW: "+ vw.getKursWert() );
            System.out.println("\t\tBMW: "+ bmw.getKursWert() );
            vw.setKursWert( vw.getKursWert() +
                           (int)Math.round( Math.random()*10 ) - 5 );
            bmw.setKursWert( bmw.getKursWert() +
                           (int)Math.round( Math.random()*10 ) - 5 );
        }
    }
}

```

Zum Erzeugen des zufälligen Kursverlaufs werden zwei statische Methoden der Klasse `Math` aus dem Standard-Paket `java.lang` der Java-Bibliothek verwendet: `random` liefert eine Zufallszahl zwischen 0.0 und 1.0 vom Typ `double`; die Methode `round` rundet eine Gleitkommazahl und liefert eine ganze Zahl vom Typ `long`. (Die Klasse `Math` stellt darüber hinaus viele Methoden zur Berechnung wichtiger mathematischer Funktionen zur Verfügung, insbesondere der trigonometrischen Funktionen.)

**Lokale und anonyme Klassen.** In Abschn. 2.2.2.1 haben wir gesehen, dass Klassen in Java als Komponenten anderer Klassen deklariert werden können. In ähnlicher Weise lassen sich innere Klassen innerhalb von Anweisungsblöcken deklarieren. Derartige Klassen nennt man *lokale Klassen*. In diesem Absatz geht es vor allem um die Einführung von lokalen Klassen ohne Namen, sogenannten *anonymen Klassen*, und deren Beziehung zu Schnittstellen.

*lokale Klasse*

*anonyme  
Klasse*

Eine Klasse sollte immer dann als lokal deklariert werden, wenn sie einen geringen Umfang hat und nur innerhalb eines Anweisungsblocks gebraucht wird. Dies ist oft dann der Fall, wenn man nur ein Objekt von einer Klasse erzeugen möchte. Typische Beispiele werden wir im Zusammenhang mit der Implementierung von graphischen Bedienoberflächen in Kap. 5 kennen lernen. Hier benutzen wir obiges Börsenbeispiel zur Illustration. Anstatt die Klassen `Boersianer1` und `Boersianer2` paketweit zugreifbar zu machen, könnte man sie als lokale Klassen der Methode `main` deklarieren:

```
public class Main2 {
    public static void main(String argv[]){
        Aktie vw    = new Aktie( "VW",    354 );
        Aktie bmw   = new Aktie( "BMW",   548 );

        class Boersianer1 implements Beobachter {
            ... // wie oben
        }
        Beobachter peter = new Boersianer1();
        vw .anmeldenBeobachter( peter );

        class Boersianer2 implements Beobachter {
            ... // wie oben
        }
        Beobachter georg = new Boersianer2();
        vw .anmeldenBeobachter( georg );
        bmw.anmeldenBeobachter( georg );
        ...
    }
}
```

Für den Fall, dass nur einmal eine Instanz von einer Klasse erzeugt werden soll, bietet Java eine noch knappere Syntax an, mit der man Klassen deklarieren kann, ohne ihnen einen Namen zu geben. Dazu wurden Teile der Syntax der Objekterzeugung mit Teilen der Syntax von Klassendeklarationen verschmolzen. Dies kann man gut erkennen, wenn man die Klassen `Boersianer1` und `Boersianer2` als anonyme Klassen realisiert:

```
public class Main3 {
    public static void main(String argv[]){
        Aktie vw    = new Aktie( "VW",    354 );
        Aktie bmw   = new Aktie( "BMW",   548 );

        Beobachter peter = new Beobachter() {
            ... // wie der Klassenrumpf von Boersianer1
        };
        vw .anmeldenBeobachter( peter );

        Beobachter georg = new Beobachter() {
            ... // wie der Klassenrumpf von Boersianer2
        };
        vw .anmeldenBeobachter( georg );
        bmw.anmeldenBeobachter( georg );
        ...
    }
}
```

Die Deklarationen der Variablen `peter` und `georg` bleiben gleich. Ihnen wird jeweils ein Objekt der anonymen Klasse zugewiesen, die auf der rechten Seite der Zuweisung deklariert ist. Anstelle des Konstruktornamens er-



scheint der Name der implementierten Schnittstelle. Dadurch wird insbesondere festgelegt, welche öffentlichen Methoden die anonyme Klasse zur Verfügung stellt und was ihr Supertyp ist. Wie bei anderen inneren Klassen können in lokalen Klassen alle sichtbaren Programmelemente des umfassenden Programmkontextes benutzt werden, also insbesondere die Attribute der umfassenden Klassendeklaration.

Im obigen Beispiel wurde ein Schnittstellentyp verwendet, um den Supertyp einer anonymen Klasse festzulegen. Ganz entsprechend kann man auch einen Klassentyp  $K$  als Supertyp einer anonymen Klasse angeben. Dann erbt die anonyme Klasse von der Klasse  $K$ . Anwendungen dieser Form anonymer Klassen werden wir in Kap. 5 studieren.

## Selbsttestaufgaben

### Aufgabe 1: Die *ist-ein*-Beziehung

- a) Der Programmierer J. Ava soll bei der objektorientierten Implementierung eines Computeralgebrasystems helfen. Die Zahlentypen  $\mathbb{R}$  (reelle Zahlen),  $\mathbb{Q}$  (rationale Zahlen),  $\mathbb{N}$  (natürliche Zahlen),  $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ ,  $\mathbb{R}_{\geq 0} = \{r \in \mathbb{R} \mid r \geq 0\}$  und  $\mathbb{Z}$  (ganze Zahlen) sollen dabei in geeigneter Weise durch Schnittstellentypen in Java ausgedrückt werden.

**Aufgabe:** Deklarieren Sie für jeden Zahlentyp einen Schnittstellentyp mit leerer Methodenliste. Nutzen Sie dabei sinnvoll Subtyping, um die *ist-ein*-Beziehung zwischen den Zahlen zu realisieren!

- b) Kommentieren Sie die folgende Aussage! Berücksichtigen Sie bei Ihrem Kommentar die Eigenschaften der *ist-ein*-Beziehung:

“Sei S eine Subklasse einer Klasse T. Da man zu S noch Methoden und Attribute hinzufügen kann, besitzen Objekte vom Typ S immer mehr Eigenschaften als Objekte vom Typ T. Z.B. kann man von einer Klasse Fahrrad sinnvoll eine Klasse Fahrzeug ableiten, da die Klasse Fahrzeug ja mehrere Fortbewegungsmittel beschreibt. Fahrrad könnte nur eine Methode *treten* und Fahrzeug zusätzlich noch die Methoden *schwimmen* und *segeln* besitzen, da Fahrzeug ja allgemeiner ist.”

### Aufgabe 2: Klassifikation und Subtyping

Ein Programmierer soll eine Menge geometrischer Figuren wie z.B. Kreise und Rechtecke in einer gemeinsamen Liste verwalten. Die Liste muss später auch andere Figuren wie z.B. Dreiecke oder Rauten aufnehmen können.

Implementieren Sie je eine Klasse für Kreise und Rechtecke sowie eine Klasse `FigurenListe` zur Verwaltung der Figuren mit den im Folgenden beschriebenen Eigenschaften:

- Die Figurenobjekte sollen jeweils über eine Methode `anzeigen` zur Darstellung der Figur verfügen.
- Die Darstellung einer Figur kann durch Ausgabe des Textes „Kreis anzeigen“, „Rechteck anzeigen“ etc. auf der Systemausgabe simuliert werden. (Instanzvariablen wie z.B. der Kreismittelpunkt müssen nicht deklariert werden.)
- Die Klasse `FigurenListe` soll ein Attribut `figurenListe` vom Typ `LinkedList<ET>` für einen geeigneten Elementtyp `ET` enthalten. Dieser Elementtyp muss sicherstellen, dass in der Liste nur Figuren mit den angegebenen Eigenschaften gespeichert werden können.

- Die Klasse `FigurenListe` soll die Methoden `figurAnfuegen`, `letzteFigurAuslesen` und `alleAnzeigen` zur Verfügung stellen.
  - Mit Hilfe der Methode `figurAnfuegen` soll ein Figurenobjekt am Ende von `figurenListe` eingefügt werden.
  - Die Methode `letzteFigurAuslesen` soll das letzte gespeicherte Figurenobjekt zurückgeben.
  - Mit Hilfe der Methode `alleAnzeigen` können alle Elemente der Liste nacheinander am Bildschirm angezeigt werden.

Die parametrische Klasse `LinkedList` ist in dem Bibliothekspaket `java.util` enthalten und braucht nicht implementiert zu werden.

### Aufgabe 3: Typüberprüfung

Die Implementierung der unten aufgeführten `main`-Methode führt zu einem Laufzeitfehler. Erläutern Sie, wo und warum dieser Fehler auftritt.

```
class Person {
    private String name;

    public Person(String pName) {
        name = pName;
    }
    public String getName () {
        return name;
    }
}

class Angestellter extends Person {
    private String position;

    public Angestellter (String aName, String aPosition) {
        super(aName);
        position = aPosition;
    }
    public String getPosition () {
        return position;
    }
}

public class Main {
    public static void main (String[] argv) {
        Angestellter [] angestellte = new Angestellter [2];
        angestellte [0] = new Angestellter ("Wilfried Kramer",
                                           "Projektleiter");
        angestellte [1] = new Angestellter ("Heike Schippang",
                                           "SW-Entwicklerin");
        Person [] personen = angestellte;
```

```
personen [0] = new Person ("Egon Mueller");
for (int i = 0; i < 2; i++) {
    System.out.println (angestellte[i].getName());
    System.out.println (angestellte[i].getPosition());
}
}
```

#### Aufgabe 4: Aufzählungstypen

1. Implementieren Sie eine Klasse `Tag` analog zur Klasse `Wochentag` aus den Selbsttestaufgaben der Kurseinheit 2. Anstatt den Typ `int` zu benutzen, um einen Tag der Woche zu repräsentieren, verwenden Sie in dieser Aufgabe einen geeigneten Aufzählungstyp `Wochentag` als Typ für das private Attribut sowie für die Methodenparameter.

Während die Signaturen für die Methoden `naechsterTag`, `vorhergehenderTag` und `toString` unverändert bleiben, sollen die Signaturen der Methoden `setTag` und `getTag` wie folgt abgeändert werden:

```
public class Tag {
    ...
    public void setTag(Wochentag tag) { ... }
    public Wochentag getTag() { ... }
    ...
}
```

2. Beantworten Sie folgende Frage: Warum braucht die Methode `setTag` jetzt keine Ausnahme mehr zu werfen?



## Musterlösungen zu den Selbsttestaufgaben

### Aufgabe 1: Die *ist-ein*-Beziehung

- a) Die folgenden Klassen erfüllen die geforderten Eigenschaften und beschreiben eine sinnvolle Subtyphierarchie:

```
interface Real { }
interface RealGreaterZero extends Real { }
interface Rational extends Real { }
interface Integer extends Rational { }
interface NaturalWithZero extends Integer, RealGreaterZero { }
interface Natural extends NaturalWithZero { }
```

Jede reelle Zahl, die größer oder gleich Null ist, ist eine reelle Zahl. Jede rationale Zahl ist eine reelle Zahl. Jede ganze Zahl ist auch eine rationale Zahl. Jede natürliche Zahl oder Null ist auch eine reelle Zahl, die größer gleich Null ist, und eine ganze Zahl. Und schließlich ist jede natürliche Zahl auch eine Zahl aus  $\mathbb{N}_0$ .

- b) Die Aussage macht natürlich keinen Sinn, denn von einem Subtypobjekt wird verlangt, dass man es überall dort verwenden kann, wo man auch ein Supertypobjekt verwendet. Bei dem gegebenen Beispiel müsste Fahrzeug eine Superklasse von Fahrrad sein. Methoden wie schwimmen oder segeln sollten dann bei den Subklassen von Fahrzeug untergebracht werden, deren Objekte auch wirklich schwimmen oder segeln können. In einer Superklasse darf man nur Methoden unterbringen, die alle ihre Subklassen gemeinsam haben. Z.B. könnte eine solche Methode in Fahrzeug die Höchstgeschwindigkeit eines Fahrzeuges als Ergebnis liefern. Objekte von Subklassen sind somit spezieller und nicht allgemeiner als die ihrer Superklassen.

### Aufgabe 2: Klassifikation und Subtyping

Die allen Figuren gemeinsamen Eigenschaften (hier ist es nur eine, nämlich anzeigen) werden in der Schnittstelle `Figur` zusammengefasst. Wenn alle Klassen, die Figuren darstellen wie `Kreis`, `Rechteck` etc., diese Schnittstelle implementieren, sind sie damit Subtypen von `Figur`. Wird als Elementtyp für `LinkedList` `Figur` gewählt, können Objekte all dieser Klassen als Elemente verwendet werden. Die Klasse `FigurenListe` verwendet als Parameter- und Rückgabetypen für ihre Methoden ebenfalls den Typ `Figur`. Daher kann die Klasse `FigurenListe` beliebige Figurenobjekte verwalten.

Folgende Implementierung löst die gestellte Aufgabe :

```

public interface Figur {
    public void anzeigen ();
}

public class Kreis implements Figur {
    // Deklaration der Instanzdaten wie
    // Kreismittelpunkt und Radius
    public void anzeigen () {
        System.out.println( "Kreis anzeigen");
    }
}

public class Rechteck implements Figur {
    // Deklaration der Instanzdaten wie
    // linke obere Ecke, Breite, Hoehe
    public void anzeigen () {
        System.out.println( "Rechteck anzeigen");
    }
}

public class FigurenListe {
    private java.util.LinkedList<Figur> figurenListe;

    public FigurenListe() {
        figurenListe = new java.util.LinkedList<Figur>();
    }

    public void figurAnfuegen (Figur figur) {
        figurenListe.addLast(figur);
    }
    public Figur letzteFigurAuslesen() {
        return figurenListe.getLast();
    }
    public void alleAnzeigen() {
        for (Figur figur : figurenListe)
            figur.anzeigen();
    }
}

```

### Aufgabe 3: Typüberprüfung

Zur Laufzeit wird bei der Anweisung

```
personen [0] = new Person ("Egon Mueller");
```

eine Ausnahme erzeugt. Durch die vorhergehende (korrekte) Zuweisung

```
Person [] personen = angestellte;
```

referenzieren personen und angestellte dasselbe Feldobjekt.

Würde die Anweisung `personen [0] = new Person ("Egon Mueller");` ausgeführt, könnte es in der Folge beim Zugriff auf den ersten Angestellten (`angestellte[0]`) zu Fehlern kommen, da in ihm nicht wie erwartet eine Referenz auf ein Objekt vom Typ `Angestellter` gespeichert ist, sondern eine Referenz auf ein Objekt vom Typ `Person`. Für `Person`-Objekte ist die Methode `getPosition` aber nicht definiert. Damit eine solche Situation gar nicht erst entstehen kann, wird bereits die Zuweisung eines `Person`-Objekts an ein Element von `personen` durch das Laufzeitsystem abgelehnt. Zuweisungen von Objekten des Typs `Angestellter` führen dagegen zu keinem Fehler.

#### Aufgabe 4: Aufzählungstypen

1. Die Klasse `Tag` kann analog zur Klasse `Wochentag` aus den Selbsttestaufgaben der Kurseinheit 2 wie folgt implementiert werden.

Anstatt den Typ `int` zu benutzen, um einen Tag der Woche zu repräsentieren, wird jetzt der Aufzählungstyp `Wochentag` als Typ für das private Attribut sowie für die Methodenparameter verwendet. In unten stehender Implementierung wurde das ehemalige Attribut `tag` ersetzt durch das Attribut `aktuellerTag`.

```
public class Tag {

    public enum Wochentag {
        Montag, Dienstag, Mittwoch, Donnerstag,
        Freitag, Samstag, Sonntag;
    }

    private Wochentag aktuellerTag = Wochentag.Montag;

    public void setTag(Wochentag tag) {
        aktuellerTag = tag;
    }

    public Wochentag getTag() {
        return aktuellerTag;
    }

    public void naechsterTag() {
        aktuellerTag =
            Wochentag.values()[ (aktuellerTag.ordinal() + 1) % 7 ];
    }
}
```



```
public void vorhergehenderTag() {
    int index;
    if (aktuellerTag.ordinal() == 0) {
        index = 6;
    }
    else {
        index = aktuellerTag.ordinal() - 1;
    }
    aktuellerTag = Wochentag.values()[index];
}

public String toString() {
    return aktuellerTag.toString();
}
}
```

2. Die Methode `setTag` braucht jetzt keine Ausnahme mehr zu werfen, da ihr Parametertyp `Wochentag` garantiert, dass ihr als aktuelle Parameter nur gültige Werte übergeben werden.

# Studierhinweise zur Kurseinheit 4

Diese Kurseinheit beschäftigt sich mit Abschnitt 3.3 und Kapitel 4 des Kurstextes. In ihrem Mittelpunkt steht die Behandlung von Vererbung und das Studium von objektorientierten Programmbausteinen. Sie sollten den Kurstext zu dieser Kurseinheit im Detail studieren und verstehen. Versuchen Sie, an eigenen Beispielen Vererbung einzusetzen. Prüfen Sie nach dem Durcharbeiten des Kurstextes, ob Sie die Lernziele erreicht haben. Arbeiten Sie dazu auch wieder die Selbsttestaufgaben am Ende der Kurseinheit durch.

## **Lernziele:**

- Sprachmittel für Vererbung, insbesondere Erweitern ererbter Programmteile und Überschreiben von Methoden.
- Einsatz von Vererbung zur Spezialisierung von Klassen.
- Subclassing: Zusammenspiel von Subtyping und Vererbung, abstrakte Klassen.
- Unterschied zwischen einfacher und mehrfacher Vererbung.
- Kapselungsaspekte im Zusammenhang mit Vererbung.
- Auflösen von Methodenaufrufen, insbesondere im Zusammenspiel mit dem Überladen, sowie Vererben und Überschreiben von Methoden.
- Verstecken von Attributen vs. Überschreiben von Methoden: Gemeinsamkeiten und Unterschiede.
- Vor- und Nachteile objektorientierter Programmierung für die Wiederverwendung.
- Techniken zur Realisierung objektorientierter Programmbausteine.

**Lernziele (Fortsetzung):**

- Anwendung von Klassifikation am Beispiel der Ausnahmebehandlung in Java.
- Abstraktion und Spezialisierung anhand der Stromklassen von Java.
- Programmierung mit Strömen.

## 3.3 Vererbung

Vererbung ermöglicht es, aus existierenden Klassen durch Erweiterung und Veränderung neue, speziellere Klassen zu bauen. Dieser Abschnitt stellt die dafür notwendigen Sprachmittel vor und beschreibt deren Anwendung. Danach geht er detaillierter auf die Beziehung zwischen Vererbung und Subtyping ein und erläutert schließlich die im Zusammenhang mit Vererbung relevanten Kapselungsmechanismen.

### 3.3.1 Vererbung: Das Sprachkonzept und seine Anwendung

Vererbung ist ein zentrales Sprachkonzept objektorientierter Programmiersprachen. Dieser Abschnitt bietet dazu eine dreiteilige Einführung. Zunächst erläutert er, was Vererbung von Programmteilen im engeren Sinne bedeutet, dann zeigt er, wie ererbte Methoden angepasst werden können, und schließlich demonstriert er die Anwendung von Vererbung zur Spezialisierung von Klassen.

#### 3.3.1.1 Vererbung von Programmteilen

*Vererbung* (engl. *inheritance*) bedeutet im engeren Sinne, dass eine Klasse Programmteile von einer anderen Klasse automatisch übernimmt. In Java ist Vererbung immer mit Subtyping verbunden (siehe Abschnitt 3.3.2) und wird dementsprechend gemeinsam deklariert: *Subklasse* *extends* *Superklasse* (vgl. die Seiten 155 und 160). Am Beispiel von Abb. 3.9, in dem die Klasse *Subklasse* von *Superklasse* erbt, werden wir die Vererbung der unterschiedlichen Komponenten einer Klasse erläutern.

*Vererbung im engeren Sinne*

Grundsätzlich werden alle Attribute und Methoden der Superklasse an die Subklasse vererbt, insbesondere auch die statischen Attribute und Methoden (vgl. aber Abschn. 3.3.3). *Subklasse* besitzt also die beiden Attribute *a* und *b* sowie die Methode *m*. Konstruktoren werden nicht vererbt. Wenn eine Subklasse keine eigenen Konstruktoren deklariert, steht zur Erzeugung von Subklassen-Objekten demnach nur der default-Konstruktor zur Verfügung (vgl. S. 80).

Als erste Anweisung kann ein Subklassen-Konstruktor einen Superklassen-Konstruktor mittels der Syntax `super (Parameter)` aufrufen; dies ist in der mit (\*) markierten Zeile illustriert. Fehlt eine solche Anweisung, wird am Anfang der Ausführung eines Konstruktors automatisch der parameterlose Konstruktor der Superklasse aufgerufen. Dadurch können in der Superklasse deklarierte Attribute initialisiert werden, auch wenn die Subklasse z.B. durch Verwendung des Modifiers `private` keinen Zugriff auf diese Attribute hat. Für das Beispiel bedeutet das, dass der Aufruf des Konstruktors `Subklasse()` in der ersten Zeile der Methode `main` implizit zu einem Aufruf von `Superklasse()` führt, das Attribut *a* des neu erzeugten Objekts also

```

class Superklasse {
    String a;
    int    b;

    Superklasse(){ a = "Urin";    }
    Superklasse( int i ){ b = i; }

    void m(){ System.out.println("stinkt"); }
}

class Subklasse extends Superklasse {
    Subklasse(){ }

    Subklasse( int i, int j ){
    (*)    super(i+j);
        a = "nicht notwendig";
    }
}

class Vererbungstest {
    public static void main( String[] args ){
        Subklasse sk = new Subklasse();
        System.out.print( sk.a );
        sk.m();
    }
}

```

Abbildung 3.9: Vererbung von einer Super- zu einer Subklasse

entsprechend initialisiert wird. Insgesamt gibt die Methode `main` demnach das Wort „Urinstinkt“ aus.

Außer Attributen und Methoden werden auch innere Klassen vererbt.

### 3.3.1.2 Erweitern und Anpassen von Ererbtem

Vererbung geht üblicherweise Hand in Hand mit dem Erweitern und Anpassen der ererbten Implementierungsteile. Wir behandeln zunächst das Hinzufügen von neuen Implementierungsteilen (neue Felder und Methoden) was wir im Folgenden *Erweitern* nennen wollen und daran anschließend das Anpassen ererbter Methoden. Dann erläutern wir Vererbung im Zusammenhang mit inneren Klassen. In diesem Abschnitt liegt unser Fokus auf den *Techniken*, die für Erweiterungen und Anpassungen zur Verfügung stehen, während in Abschnitt 3.3.1.3 der *Zweck* der Anpassung von Ererbtem im Mittelpunkt steht und die *Eignung* von Programmcode für Spezialisierung.

**Erweitern und Überschreiben.** Als Beispiel für das Erweitern und Anpassen von Implementierungen betrachten wir die Klassen `Person` und `Student` aus Abschn. 1.3.2:

```

class Person {
    String name;
    int geburtsdatum; /* in der Form JJJJMMTT */

    Person( String n, int gd ) {
        name = n;
        geburtsdatum = gd;
    }
    void drucken() {
        System.out.println("Name: "+ this.name);
        System.out.println("Geburtsdatum: "+ geburtsdatum);
    }
    boolean hat_geburtstag ( int datum ) {
        return (geburtsdatum % 10000) == (datum % 10000);
    }
}

class Student extends Person {
    int matrikelnr;
    int semester;

    Student( String n, int gd, int mnr, int sem ) {
        super( n, gd );
        matrikelnr = mnr;
        semester = sem;
    }
    void drucken() {
        (*) super.drucken();
        System.out.println( "Matrikelnr: " + matrikelnr );
        System.out.println( "Semesterzahl: " + semester );
    }
    void semesterWeiterschalten() {
        semester = semester + 1;
    }
}

```

Die Klasse `Student` erbt die Attribute `name` und `geburtsdatum` sowie die Methode `hat_geburtstag` von der Klasse `Person`. Sie erweitert die Klasse `Person` um die Attribute `matrikelnr` und `semester` sowie um die Methode `semesterWeiterschalten`, die beim Wechsel in das nächste Semester aufgerufen wird. Die Methode `drucken` der Klasse `Person` wird in der Klasse `Student` an die neuen Bedürfnisse dieser Klasse angepasst.

Jede objektorientierte Programmiersprache ermöglicht es, Methoden einer Superklasse den Bedürfnissen der Subklasse anzupassen. In Java gibt es dafür die folgende Möglichkeit: Superklassen-Methoden können in der Subklasse durch eine Methode gleicher Signatur ersetzt werden; man sagt dann, dass

Über  
schreiben

die (neue) Subklassen-Methode die Superklassen-Methode *überschreibt* (im Englischen spricht man von *overriding*, was im Allgemeinen „aufheben“, „außer Kraft setzen“ bedeutet). Die überschreibende Methode kann eine komplett neue Implementierung besitzen. Häufig möchte man aber in der überschreibenden Methode auch auf die Implementierung der entsprechenden Methode der Superklasse zurückgreifen können. Dies ist in Java durch Voranstellen des Schlüsselworts `super` möglich<sup>15</sup>. Die Methode *Methodenname* der Superklasse kann also mittels `super.Methodenname(Parameter)` aus dem Rumpf der überschreibenden Methode aufgerufen werden. Als impliziter Parameter wird dabei das aktuelle `this`-Objekt übergeben.

Die Methode `drucken` der Klasse `Student` illustriert das Überschreiben und Nutzen von Superklassenmethoden: Sie besitzt dieselbe Signatur wie die Methode `drucken` ihrer Superklasse. Sie überschreibt die Methode `drucken` der Klasse `Person`, um auch die zusätzlichen Attribute von `Student` zu drucken. Dabei bedient sie sich der Methode `drucken` ihrer Superklasse, um die von `Person` geerbten Attribute zu drucken. Die mit (\*) markierte Zeile zeigt die dafür in Java vorgesehene, bereits oben erwähnte Aufrufsyntax.

Bei Methodenaufrufen mit `super` lässt sich immer zur Übersetzungszeit feststellen, welche Methode auszuführen ist: Enthält die Superklasse `SK` derjenigen Klasse, in der der `super`-Aufruf steht, eine passende Methodendeklaration, wähle diese Deklaration; ansonsten suche in der Superklasse von `SK` bzw. in deren Superklassen nach einer passenden Methodendeklaration. Methodenaufrufe mit `super` lassen sich also statisch binden.

**Vererbung und innere Klassen.** Außer Attributen und Methoden werden auch innere Klassen vererbt. Genauso wie Subklassen um zusätzliche Attribute erweitert werden können, können ihnen auch weitere innere Klassen hinzugefügt werden. Diese neuen inneren Klassen können wieder von anderen Klassen erben. Interessant ist insbesondere der Fall, in dem eine neue innere Klasse von einer anderen inneren Klasse erbt, die von der Superklasse der umfassenden Klasse geerbt wurde.

Dieses Szenario, bei dem also die Spezialisierung der umfassenden Klasse mit der Spezialisierung einer ihrer inneren Klassen einhergeht, wollen wir an einem kleinen Beispiel studieren. Dazu betrachten wir die Klasse `LinkedList` mit der inneren Klasse `ListIterator` aus Kap. 2, Seite 119. Da wir die erweiterte Klasse im Folgenden für beliebige Objekte nutzen wollen, gehen wir davon aus, dass die Listenelemente vom Typ `Object` sind. Im Übrigen müssen wir den Zugriff auf die Attribute innerhalb von Subklassen gestatten. Dazu werden die Attribute als *geschützt* (engl. *protected*) deklariert. (Genaueres zu Kapselungsaspekten im Zusammenhang mit Vererbung

geschützte  
Attribute

<sup>15</sup>`super` bezeichnet kein neues Objekt, sondern weist den Compiler lediglich an, wo er nach der aufzurufenden Methode suchen soll.

```

public class LinkedList {
    protected Entry header = new Entry(null, null, null);
    protected int size = 0;

    public LinkedList() { ... }
    public Object getLast() { ... }
    public Object removeLast() { ... }
    public void addLast(Object e) { ... }
    public int size() { ... }

    static class Entry {
        Object element;
        Entry next;
        Entry previous;
        Entry(Object e, Entry n, Entry p) { ... }
    }

    public ListIterator listIterator() { ... }

    public class ListIterator {
        protected int nextIndex = 0;
        protected Entry next = header.next;

        public boolean hasNext() { ... }
        public Object next() { ... }
    }
}

```

Abbildung 3.10: Die überarbeitete Klasse `LinkedList`

erläutert Abschn. 3.3.3.) Abbildung 3.10 bietet eine Zusammenstellung der so überarbeiteten Klasse `LinkedList`.

Die Iteratoren der Klasse `LinkedList` erlauben es nur, die Listen in einer Richtung zu durchlaufen.

Die erweiterte Listenklasse `ExtendedList` soll diese Einschränkung aufheben und es gestatten, die aktuelle Position ans Listenende zu setzen. Darüber hinaus soll es ermöglicht werden, eine Liste hinter der aktuellen Position des Iterators abzuschneiden, d.h. alle Elemente hinter der Position aus der Liste zu entfernen. Dazu erweitert die Klasse `ExtendedList` die von der Klasse `LinkedList` geerbte Klasse `ListIterator` und stellt eine neue Methode `extListIterator` zur Verfügung, die Objekte der mächtigeren Iteratorklasse liefert (siehe Abb. 3.11). Zur Verdeutlichung sei explizit darauf aufmerksam gemacht, dass die Klasse `ExtendedList` insgesamt drei innere Klassen besitzt: die geerbten Klassen `Entry` und `ListIterator` sowie die neu hinzugefügte Klasse `ExtListIterator`.



```

public class ExtendedList extends LinkedList {

    public ExtListIterator extListIterator() {
        return new ExtListIterator();
    }

    public class ExtListIterator extends ListIterator {

        public boolean hasPrevious() { return nextIndex != 0; }

        public Object previous() {
            if( nextIndex==0 ) throw new NoSuchElementException();
            next = next.previous;
            nextIndex--;
            return next.element;
        }
        public void setToEnd() {
            next = header;
            nextIndex = size;
        }
        public void cut() {
            if( next != header ) {
                // es existieren zu entfernende Elemente
                header.previous = next.previous;
                next.previous.next = header;
                if( nextIndex == 0 ) header.next = header;
                size = nextIndex;
                next = header; // Iterator ans Ende setzen
            } // wenn next == header, ist nichts zu tun
        }
    }
}

```

Abbildung 3.11: Die Klasse ExtendedList

Eine Anwendung der erweiterten Listenklasse werden wir im folgenden Unterabschnitt kennen lernen.

### 3.3.1.3 Spezialisieren mit Vererbung

In diesem Abschnitt steht der *Zweck* der Anpassung von Ererbtem im Mittelpunkt, nämlich der Anpassung existierenden Programmcodes an speziellere Erfordernisse sowie die *Eignung* von Programmcode für Spezialisierung. (Spezialisierung wird hier zunächst unabhängig von einer Subtyphierarchie betrachtet.)

Die Spezialisierung existierender Klassen ist eine der zentralen objektorientierten Programmierstechniken. Typische Anwendung der Spezialisierung ist die Anpassung eines allgemeinen Implementierungsrahmens (etwa zur Oberflächenprogrammierung) an spezielle Erfordernisse oder die Anpassung

bestimmter Klassen eines existierenden Programmsystems an neue Rahmenbedingungen. In Abschn. 3.1, S. 153, haben wir bereits an einem kleinen Beispiel gezeigt, wie Vererbung zur Spezialisierung genutzt werden kann. Hier wollen wir weitere Aspekte anhand einer Spezialisierung der Browser-Klasse von Abb. 2.6, S. 97, studieren, die uns erste Hinweise darauf gibt, welche Art von Programmcode für Spezialisierung geeignet ist und welche nicht.

Anschließend diskutieren wir kurz, was zu beachten ist, wenn Programme mit dem Ziel erstellt werden, dass von ihnen geerbt werden soll.

**Spezialisieren der Browser-Klasse.** Realistische Internet-Browser verwalten für jedes Browser-Fenster eine Liste der angezeigten WWW-Seiten und stellen die Operationen *vor* und *zurück* zur Verfügung, mit denen man in der Liste vor- und zurücklaufen kann: Zurücklaufen bedeutet dabei, dass man zu den in der Betrachtungsreihenfolge zurückliegenden Seiten geht, Vorlaufen bedeutet, dass man sich Seiten anzeigen lässt, die man das erste Mal nach der aktuellen Seite geladen hat. Steht man inmitten der Seitenliste und lädt eine neue Seite, werden alle durch Vorlaufen erreichbaren Seiten aus der Liste entfernt und die neue Seite an die verbleibende Liste angehängt.

```
class VorZukBrowser extends Browser {
    ExtendedList vorzurueckliste = new ExtendedList();
    ExtendedList.ExtListIterator seitenIter =
        vorzurueckliste.extListIterator();

    // Konstruktor zum Starten des ersten Browsers
    VorZukBrowser( W3Server server ) {
        super( server );
    }

    // Konstruktor zum Starten weiterer Browser
    VorZukBrowser() { }

    static void start( W3Server server ) {
        VorZukBrowser vzb = new VorZukBrowser(server);
        vzb.vorzurueckliste.addLast( startseite );
        vzb.seitenIter.setToEnd();
    (*) VorZukBrowser.interaktiveSteuerung();
    }

    static void interaktiveSteuerung() {
        ... /* siehe folgende Abbildung */
    }
}
```

Abbildung 3.12: Eine Spezialisierung der Browser-Klasse

Wir wollen unseren rudimentären Browser aus Abschn. 2.1.4 mit einer solchen „vor-zurück“-Liste ausstatten und nennen die so spezialisierte

Browser-Klasse `VorZukBrowser` (siehe Abb. 3.12). Ein `VorZuk-Browser` ist ein Browser, der eine „vor-zurück“-Liste mit einem entsprechenden Iterator besitzt. Zur Verwaltung der „vor-zurück“-Liste benutzen wir die Klasse `ExtendedList` des vorigen Unterabschnitts. Die Klasse `Browser` wird um zwei Attribute erweitert und mit passenden Konstruktoren versehen. Bei der Deklaration des einstelligen Konstruktors muss der Aufruf des entsprechenden Konstruktors in der Superklasse explizit angegeben werden, beim nullstelligen Konstruktor<sup>16</sup> geschieht dieser Aufruf implizit (s. S. 217). In beiden Fällen ist der entsprechende Browser-Konstruktor auch für `VorZukBrowser`-Objekte ausreichend.

Während die geerbten Methode `laden` und `initialisieren` unverändert übernommen werden können, müssen die Methoden `start` und `interaktiveSteuerung` durch vollständig neue Versionen ersetzt werden, um den neuen Anforderungen gerecht zu werden. Es reicht in diesem Fall nicht aus, die Superklassenmethoden nur am Anfang oder Ende zu ergänzen, wie dies bei der Methode `drucken` in der Klasse `Student` der Fall war (vgl. S. 218). Sehr schön lässt sich das anhand der Methode `interaktiveSteuerung` studieren, deren Fassung in der Subklasse im Wesentlichen durch Einfügen zusätzlicher Anweisungen aus der Fassung in der Superklasse entstand. Insbesondere werden in der erweiterten Fassung die Steuerzeichen *v* und *z* unterstützt, mit denen man in den „vor-zurück“-Listen einen Schritt vor- bzw. zurückgehen kann (s. Abb. 3.13). Die Implementierung der Methode `interaktiveSteuerung` der Klasse `VorZukBrowser` kann nicht durch Vererbung von der bereits vorhandenen Implementierung in der Superklasse profitieren.

**Programmierung für Vererbung.** Wie gut sich Klassen bzw. Methoden für Vererbung eignen, hängt sowohl von den verwendeten Programmier-techniken als auch von der Strukturierung der Klassen und Methoden ab. Beispielsweise sind zentral organisierte Steuerungen wenig für die Spezialisierung mittels Vererbung geeignet, wie wir an Hand der Spezialisierung der Klasse `Browser` illustriert haben. In der Subklasse `VorZukBrowser` musste die Methode `interaktiveSteuerung` z.B. komplett neu geschrieben werden. Die entsprechende Methode der Superklasse konnte nicht wiederverwendet werden. Im Vergleich dazu ermöglichen ereignisorientierte Programmier-techniken, die wir im Zusammenhang mit der Steuerung graphischer Bedienoberflächen in Kap. 5 kennen lernen werden, Programmstrukturen, die sich erheblich besser mittels Vererbung erweitern und anpassen lassen.

Bei der Eignung für Vererbung spielt auch die geeignete Aufteilung des Programmcodes auf die Methoden eine wichtige Rolle. Grundsätzlich bie-

---

<sup>16</sup>Man beachte, dass die Deklaration des nullstelligen Konstruktors notwendig ist, da Konstruktoren nicht vererbt werden und die Existenz des einstelligen Konstruktors dazu führt, dass kein default-Konstruktor bereitgestellt wird (vgl. S. 80).

```

static void interaktiveSteuerung() {
    char steuerzeichen = '0';
    do {
        Konsole.writeString("Steuerzeichen eingeben [lnwvze]: ");
        try {
            String eingabe = Konsole.readString();
            ... // wie in Klasse Browser
        }
        VorZukBrowser aktBrws =
            (VorZukBrowser) gestarteteBrowser[aktBrowserIndex];
        switch( steuerzeichen ){
        case 'l':
            Konsole.writeString("Seitenadresse eingeben: ");
            String seitenadr = Konsole.readString();
            W3Seite neueSeite = meinServer.holenSeite( seitenadr );
            aktBrws.laden( neueSeite );
            aktBrws.seitenIter.cut();
            aktBrws.vorzurueckliste.addLast( neueSeite );
            aktBrws.seitenIter.setToEnd();
            break;
        case 'v':
            if( aktBrws.seitenIter.hasNext() ){
                aktBrws.laden( (W3Seite)aktBrws.seitenIter.next() );
            }
            break;
        case 'z':
            if( aktBrws.seitenIter.hasPrevious() ){
                // Iterator vor die aktuelle Seite stellen
                aktBrws.seitenIter.previous();
                if( aktBrws.seitenIter.hasPrevious() ){
                    aktBrws.laden( (W3Seite)aktBrws.seitenIter.previous() );
                }
                // Iterator hinter die aktuelle Seite stellen
                aktBrws.seitenIter.next();
            }
            break;
        case 'n':
            new VorZukBrowser();
            break;
        case 'w':
            aktBrowserIndex=(aktBrowserIndex+1)%naechsterFreierIndex;
            break;
        case 'e':
            System.exit( 0 );
        default:
            Konsole.writeString("undefiniertes Steuerzeichen\n");
        }
    } while( true );
}

```

Abbildung 3.13: Browsersteuerung mit Vor-Zurück-Liste

ten mehrere kleine Methoden mehr Flexibilität für die Vererbung als wenige große (andererseits schaden zu viele Methoden der Lesbarkeit). Geeignete Aufteilung ist aber nicht nur eine Frage der Größe. Hätten wir beispielsweise keine spezielle `start`-Methode für `Browser` vorgesehen und die Methode `interaktiveSteuerung` direkt im Konstruktor aufgerufen (wie dies im ersten `Browser`-Entwurf auf S. 86 der Fall war), hätten wir beim einstelligen Konstruktor für die Klasse `VorZukBrowser` den Programmcode der Superklasse `Browser` nicht wieder verwenden können, da in dem Fall der Konstruktor `Browser` die falsche Methode zur interaktiven Steuerung aufrufen würde. Durch Benutzung der `start`-Methode konnte diese Änderung auf eine spezifische Stelle konzentriert werden.

Zwei weitere Aspekte bedürfen beim Programmieren für Vererbung besonderer Beachtung.

- Klassenmethoden sollten mit Bedacht verwendet werden. Da sie keiner dynamischen Bindung unterliegen (siehe Abschnitt 2.1.4.2), muss jeder Aufruf einer Klassenmethode, die in der Subklasse neu definiert wurde, explizit in der Subklasse angepasst werden. In der Zeile (\*) der Abbildung 3.12 musste der ehemalige Aufruf `Browser.interaktiveSteuerung()` z.B. ersetzt werden durch `VorZukBrowser.interaktiveSteuerung()`<sup>17</sup>.
- Der Aufruf von Methoden innerhalb von Konstruktoren stellt im Zusammenhang mit Vererbung eine gewisse Gefahrenquelle dar, wie das fehlerhafte Programmfragment von Abb. 3.14 veranschaulicht.

Nehmen wir an, der Konstruktor `Unterklasse` wird bei der Erzeugung eines `Unterklasse`-Objektes `X` aufgerufen (zur Ausführungssemantik der Objekterzeugung vgl. S. 80). Dies führt zum (hier impliziten) Aufruf des Konstruktors `Oberklasse` (zur Ausführungssemantik von Konstruktoren vgl. S. 217), wobei `X` als impliziter Parameter übergeben wird. Die Ausführung des Konstruktors `Oberklasse` initialisiert das Attribut `a` von `X` und ruft dann auf diesem Objekt die Methode `m` auf (siehe die mit (\*) markierte Zeile). Wegen dynamischer Bindung führt dieser Aufruf zur Ausführung der Methode `m` aus der Unterklasse. Da zu diesem Zeitpunkt das Attribut `b` noch nicht initialisiert ist, terminiert der Aufruf `b.length()` abrupt mit einer `NullPointerException`-Ausnahme. Zusammengefasst besteht die Gefahrenquelle also darin, dass ein Methodenaufruf in einem Superklassen-Konstruktor durch dynamische Bindung zur Ausführung einer Subklassen-Methode führt, bevor der Subklassen-Konstruktor die Initialisierung der Attribute abschließen konnte. Gerade bei Spezialisierung nicht selbst verfasster Klassen kommen derartige oder ähnlich gelagerte Fehler nicht selten vor. Deshalb ist es wichtig, sich dieser Gefahr bewusst zu sein.

<sup>17</sup> Ab Java 5.0 gibt es den sogenannten *static import*, der es erlaubt, statische Methoden einer Klasse zu verwenden, ohne ihr den Namen der Klasse voranzustellen. Näheres hierzu kann der Java-Dokumentation entnommen werden.

```

class Oberklasse {
    String a;

    Oberklasse() {
        a = "aha";
    }
    (*) m();
    void m() {
        System.out.print("Laenge von a:" + a.length() );
    }
}

class Unterklasse extends Oberklasse {
    String b;

    Unterklasse() {
        b = "boff";
        m();
    }
    void m() {
        System.out.print("Laenge von b:" + b.length() );
    }
}

class KonstruktorProblemTest {
    public static void main( String[] args ) {
        new Unterklasse();
    }
}

```

Abbildung 3.14: Fehlerhafter Umgang mit Methoden in Konstruktoren

Ein weiterer wichtiger Aspekt beim Programmieren für Vererbung ist der geeignete Einsatz von Kapselungstechniken. Subklassen brauchen häufig Zugriff auf ererbte Attribute und Methoden. Deshalb muss eine Klasse, von der andere erben sollen, festlegen, welche ihrer nicht öffentlichen Attribute und Methoden in Subklassen benutzt werden dürfen. Diese besondere Schnittstelle zu Subklassen wird auch als *Vererbungsschnittstelle* oder *Vererbungsinterface* bezeichnet. Sprachkonstrukte, mit denen die zu dieser Schnittstelle gehörenden Attribute und Methoden deklariert werden können, werden in Abschn. 3.3.3 behandelt.

*Vererbungs-  
schnittstelle*

Bereits die kleinen Beispiele dieses Abschnitts geben eine erste Vorstellung von den Vorzügen und Problemen, die Vererbung mit sich bringt. Einerseits stellt sie in vielen Fällen eine wesentliche Erleichterung beim Erweitern und Anpassen von Programmen dar. Andererseits erhöht sie die Komplexität der Programme und führt zu zusätzlichen Gefahrenquellen. Allgemeiner betrachtet, sind dies zwei Seiten derselben Medaille: Programme zu erstellen,

die angepasst und wiederverwendet werden können, stellt neue methodische und technische Anforderungen an den Programmentwurf und erfordert Unterstützung durch zusätzliche, ausdrucksstarke Sprachkonzepte. Diese bringen aber auch bisher nicht vorhandene Fehlerquellen mit sich.

### 3.3.2 Vererbung, Subtyping und Subclassing

Dieser Unterabschnitt untersucht den Zusammenhang zwischen Vererbung und Subtyping, erläutert, was abstrakte Klassen und abstrakte Methoden sind und geht kurz auf Mehrfachvererbung ein. Anhand einer gegebenen Typhierarchie werden verschiedene Möglichkeiten untersucht, die Typen innerhalb dieser Hierarchie als Klassen, abstrakte Klassen und Schnittstellentypen zu realisieren. Dabei werden Vor- und Nachteile der verschiedenen Varianten diskutiert.

Vererbung im Zusammenhang mit objektorientierter Programmierung wird häufig in einem sehr allgemeinen Sinn verwendet und schließt dann Subtyping und dynamische Bindung mit ein. Im engeren Sinne bedeutet Vererbung aber nur, dass es einen *Mechanismus* gibt, mit dem eine Klasse Programmteile anderer Klassen übernehmen kann. Insbesondere könnte Vererbung unabhängig von der Subtyp-Beziehung sein; d.h. eine Klasse *K* könnte von einer Klasse *VK* erben, ohne dass *K* ein Subtyp von *VK* ist<sup>18</sup>. In Java ist Vererbung immer mit Subtyping verbunden. Häufig bezeichnet man diese verbreitete Kombination von Vererbung und Subtyping auch als *Subclassing*. Da bei Subclassing die Subklasse alle Attribute und Methoden von einer Klasse erbt, die den Supertyp implementiert, erfüllt sie auch automatisch die entsprechenden syntaktischen Bedingungen für Subtypen (siehe S. 173 f).

Subclassing

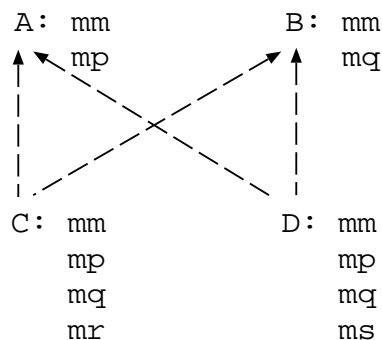


Abbildung 3.15: Die Typen A, B, C und D mit Methoden

Wie wir gesehen haben, unterstützt Java sowohl Subclassing als auch Subtyping ohne Vererbung von Implementierungsteilen. Den Zusammenhang zwischen diesen Sprachkonzepten wollen wir am Beispiel von Abb. 3.15 etwas genauer herausarbeiten. A, B, C und D seien vier Typen, die die ange-

<sup>18</sup>Eine derartige Entkopplung von Vererbung und Subtyping ist in der Sprache Sather realisiert (vgl. [SOM94]).

gebenen Methoden besitzen sollen (einfachheitshalber gehen wir davon aus, dass die Methoden keine Parameter und keine Ergebnisse haben); die Pfeile veranschaulichen die Subtyp-Beziehung zwischen den Typen. Wir betrachten drei Realisierungsvarianten:

1. Nur Subtyping: A und B werden als Schnittstellentypen realisiert, C und D als Klassen.
2. Mit einfacher Vererbung: A wird als (abstrakte) Klasse realisiert; C und D erben von A und implementieren B.
3. Mit Mehrfachvererbung: B wird ebenfalls als Klasse realisiert; C und D erben von A und B.

Der folgende Programmcode verwendet die erste Variante. Die Typen A und B sind als Schnittstellen realisiert; C und D implementieren diese Schnittstellen.

```
interface A {
    void mm();
    void mp();
}

interface B {
    void mm();
    void mq();
}

class C implements A, B {
    int a, b, c;

    public void mm(){
        c = 2000; ...
        { // dieser Block benutzt nur Attribut a und ist
          // identisch mit entsprechendem Block in Klasse D
            ... a ...
        }
        c = a + c;
    }
    public void mp(){
        // benutzt die Attribute a und c in komplexer Weise
    }
    public void mq(){
        b = 73532;
    }
    public void mr(){ ... }
}

class D implements A, B {
    int a, b;
    String d;
```



```

public void mm(){
    // dieser Block benutzt nur Attribut a und ist
    // identisch mit entsprechendem Block in Klasse C
    ... a ...
}
public void mp(){
    // benutzt die Attribute a und d in komplexer Weise
}
public void mq(){
    b = 73532;
}
public void ms(){ ... }
}

```

Der Nachteil dieser Variante ist, dass Implementierungsteile, die den Typen A und B zugeordnet werden könnten, sowohl in C als auch in D angegeben werden müssen. Im Beispiel könnte man das Attribut a und einen Teil der Implementierung von Methode `mm` dem Typ A zuordnen und das Attribut b und die Methode `mq` dem Typ B.

**Abstrakte Klassen und Methoden.** In Java können Klassen und Methoden durch Voransetzen des Schlüsselwortes `abstract` als abstrakt deklariert werden. Die Implementierung *abstrakter Klassen* muss nicht vollständig sein; d.h. in abstrakten Klassen dürfen Methoden ohne Rumpf deklariert werden. Derartige Methoden werden *abstrakte Methoden* genannt. Abstrakte Klassen stehen also zwischen Klassen und Schnittstellen: Einerseits können sie Attribute und implementierte Methoden deklarieren und diese an Subklassen vererben. Andererseits ist es nicht möglich, Objekte abstrakter Klassen zu erzeugen, und nicht für jede Methode muss eine Implementierung angegeben werden. Abstrakte Klassen können verwendet werden, um gemeinsame Implementierungsteile zukünftiger Subtypen an einer Stelle zusammenzufassen und damit die Programme kleiner und pflegeleichter zu machen.

Als Beispiel betrachten wir die zweite Realisierungsvariante der Typen A bis D. Der Typ A wird als abstrakte Klasse realisiert und erhält das Attribut a sowie die Implementierung, die `mm` in der ersten Variante von D besaß und die Teil der Implementierung von `mm` in C war; die Methode `mp`, deren Implementierung in C und D keine gemeinsamen Teile aufweist, bleibt abstrakt:

```

abstract class A {
    int a;
    public void mm() {
        // der auf 'c = 2000'; ...' folgende Block von mm
        // in C aus der ersten Variante
        // der identisch ist mit dem Rumpf von mm in D
        ... a ...
    }
    public abstract void mp();
}

```

B bleibt weiterhin als Schnittstellentyp realisiert. Die zweite Variante der Klasse C erweitert die abstrakte Klasse A und implementiert B. Im Vergleich zur ersten Variante erbt C das Attribut a und kann bei der Implementierung von mm auf die Superklassen-Implementierung zurückgreifen:

```
class C extends A implements B {
    int b, c;

    public void mm(){
        c = 2000; ...
        super.mm();
        c = a + c;
    }

    public void mp(){
        // benutzt die Attribute a und c in komplexer Weise
    }

    public void mq(){ b = 73532; }
    public void mr(){ ... }
}
```

Die zweite Variante der Klasse D erbt das Attribut a und die Implementierung von mm:

```
class D extends A implements B {
    int b;
    String d;

    public void mp(){
        // benutzt die Attribute a und d in komplexer Weise
    }
    public void mq(){ b = 73532; }
    public void ms(){ ... }
}
```

Selbstverständlich könnte man A auch zu einer vollständigen, d.h. nicht abstrakten Klasse machen, indem man eine geeignete Implementierung für mp angibt. Da sich die Implementierungen von mp in C und D unterscheiden, muss mp aus A dann in mindestens einer der Klassen C oder D überschrieben werden. Eine andere Variante wäre es, statt des Typs A den Typ B als Klasse zu realisieren.

**Mehrfachvererbung.** Wenn eine Klasse Implementierungsteile von mehreren Klassen erben kann, spricht man von *Mehrfachvererbung* (engl. *multiple inheritance*). Wenn Java Mehrfachvererbung unterstützen würde, könnten wir im Beispiel auch den Typ B als Klasse mit Attribut b und Methode mq realisieren. C und D könnten diese Implementierungsteile dann von B erben. Java erlaubt aber keine Mehrfachvererbung, hauptsächlich weil Mehrfachvererbung

*Mehrfach-  
vererbung*

zu recht unübersichtlichen und damit fehlerträchtigen Programmen führen kann.

Bei Mehrfachvererbung kann es nämlich vorkommen, dass aus verschiedenen Klassen gleich bezeichnete Implementierungsteile geerbt werden. Hätten wir beispielsweise *A* und *B* als Klassen realisiert, würden *C* und *D* zwei Implementierungen für *mm* erben. Dann wäre die Frage zu klären, welcher Implementierung der Vorzug gegeben werden soll. Werden zwei gleich benannte Attribute geerbt, muss festgelegt sein, ob beide Attribute in der Subklasse existieren sollen und wie dann beim Zugriff zwischen den Attributen unterschieden werden soll. Die Situation wird noch komplexer, wenn eine Klasse *E* von zwei Klassen *C* und *D* erbt, die eine gemeinsame Superklasse *A* besitzen. Sollen die Attribute von *A* dann mehrfach in *E* existieren oder nur einmal? Unterschiedliche Sprachen geben auf diese Fragen unterschiedliche Antworten. In C++ beispielsweise, das Mehrfachvererbung in flexibler Weise unterstützt, kann der Programmierer deklarieren, ob die Attribute von *A* in der geschilderten Situation mehrfach oder nur einfach in *E* existieren sollen.

Java verzichtet auf Mehrfachvererbung und deren Komplexität. Der Preis dafür ist, dass Aufgabenstellungen, die mit Mehrfachvererbung elegant und einfach gelöst werden könnten, in Java zu etwas aufwendigeren Programmen führen. Besitzt eine Klasse mehrere Supertypen, muss der Programmierer entscheiden, welcher Supertyp als Klasse implementiert werden soll und welche Supertypen als Schnittstelle zu realisieren sind. Mehrere Supertypen sind immer dann sinnvoll, wenn es verschiedene Kontexte für eine Klasse gibt, die jeweils eine unterschiedliche Sicht auf die Klasse benötigen. Z.B. könnte es für Objekte einer Klasse in einem Kontext nur wichtig sein, dass sie „druckbar“ sind, also z.B. die Schnittstelle *Druckbar* implementieren, während sie in einem anderen Kontext nur als serialisierbare Objekte (siehe Seite 281) benötigt werden, um sie in einer Datei abspeichern und später daraus wieder restaurieren zu können. Dafür müssen serialisierbare Objekte die Schnittstelle *Serializable* implementieren (s. auch Seite 179).

Werden auch Objekte benötigt, die genau die Methoden der Schnittstellentypen besitzen, müssen dafür zusätzliche Klassen implementiert werden. Im obigen Beispiel haben wir uns dafür entschieden, *A* als (abstrakte) Klasse zu realisieren; *B* kann also nur noch als Schnittstelle deklariert werden. Benötigen wir Objekte, die genau die Schnittstelle *B* bieten, müssen wir einen zusätzlichen Typ implementieren, den wir hier *GenauB* nennen:

```
class GenauB implements B {
    int b;
    public void mm(){ ... } // gemaess den Anforderungen von B
    public void mq(){ b = 73532; }
}
```

*GenauB*-Objekte können dann an allen Programmstellen verwendet werden, an denen in einem Programm, das mit Mehrfachvererbung realisiert wurde,

B-Objekte vorkommen. Statt Mehrfachvererbung kann in Java also nur Einfachvererbung und mehrfaches Subtyping verwendet werden. Wie man mit dieser Technik trotzdem Mehrfachvererbung simulieren kann, demonstrieren wir in Selbsttestaufgabe 3, Seite 285.

Wie wir in späteren Kapiteln sehen werden, findet die Kombination von Subclassing und Subtyping insbesondere immer da Anwendung, wo eine neue Klasse von einer selbstgeschriebenen Klasse erben soll, gleichzeitig aber auch Subtyp eines oder mehrerer Typen der Java-Bibliothek sein soll.

Ein häufig auftretender Fall ist der, dass eine selbstgeschriebenen Klasse eigentlich zusätzlich von der Klasse `Thread` erben müsste, um parallel zu anderen Programmsträngen ausgeführt werden zu können. Da das aber wegen der Einfachvererbung nicht möglich ist, bietet Java einen Umweg über die Schnittstelle `Runnable` an. Wenn eine eigenen Klasse diese Schnittstelle implementiert, kann sie die Funktionalität der Klasse `Thread` nutzen ohne von ihr zu erben. Auf dieses Problem werden wir in Kapitel 6 näher eingehen.

### 3.3.3 Vererbung und Kapselung

Kapselungskonstrukte im Sinne des Information Hiding ermöglichen es, den Zugriff auf die Implementierung eines Typs für bestimmte Benutzer einzuschränken. Die Gründe für den Einsatz von Kapselung haben wir in Abschnitt 2.2.1 erläutert. Die Kapselung kann sich auf Attribute, Konstruktoren, Methoden und Typen erstrecken. Bisher haben wir drei Arten der Zugriffsbeschränkung kennen gelernt:

1. Privater Zugriff, d.h. Zugriff nur innerhalb der Klasse, in der das entsprechende Programmelement deklariert ist.
2. Paketlokaler Zugriff, d.h. nur innerhalb des umfassenden Pakets (default access).
3. Öffentlicher Zugriff.

Bei jeder Anwendung eines Programmelements muss dann geprüft werden, ob der Zugriff entsprechend der deklarierten Zugriffsrechte erlaubt ist.

Dynamische Methodenbindung erschwert den Entwurf und die Prüfung der Zugriffsregeln. Bei einem Aufruf ist zur Übersetzungszeit nicht bekannt, welche Methode ausgeführt wird. Um Zugriffsbeschränkung trotzdem statisch prüfen zu können, verlangt man grundsätzlich, dass eine Methode, die eine andere überschreibt oder eine in einer Schnittstelle oder abstrakten Klasse deklarierte Methode implementiert, einen mindestens so großen Zugriffsbereich hat wie die überschriebene bzw. deklarierte Methode. Am einfachsten ist dies in Java für die Methoden geregelt, die in Schnittstellentypen vereinbart wurden. Solche Methoden und alle sie implementierenden Metho-

den müssen als öffentlich deklariert sein. Komplexer wird eine saubere Regelung der Kapselung im Zusammenhang mit Vererbung. Dieser Abschnitt geht auf diese Problematik kurz ein und stellt eine weitere Art der Zugriffsbeschränkung vor.

In Programmiersprachen, die Kapselung und Vererbung unterstützen, stellen sich die beiden folgenden Fragen:

- Braucht man zusätzliche Sprachkonstrukte zur Kapselung, wenn Vererbung möglich ist?
- Wie werden private Programmelemente bei der Vererbung behandelt?

Wir werden uns im Folgenden anschauen, welche Antworten Java auf diese Fragen gibt.

### 3.3.3.1 Kapselungskonstrukte im Zusammenhang mit Vererbung

Im Zusammenhang mit Vererbung behandeln wir zwei Kapselungstechniken: die Begrenzung des Zugriffsbereichs auf die Subklassenhierarchie und die Einschränkung von Vererbung und Überschreiben.

**Geschützter Zugriff.** Es gibt im Wesentlichen zwei Arten, eine Klasse zu benutzen. Man kann eine Klasse so anwenden, wie sie ist (*Anwendungsnutzung*). Oder man kann von einer Klasse erben, um eine speziellere Klasse zu entwickeln (*Vererbungsnutzung*). Während die öffentliche Klassenschnittstelle für eine direkte Anwendung ausreicht, benötigt die Subklasse häufig Zugriff auf Programmelemente, insbesondere Attribute, die man nicht öffentlich zugreifbar machen möchte. Auf diese Unterscheidung sind wir bereits bei der Klasse `LinkedList` von Abb. 3.10 gestoßen: Die Attribute `header` und `size` sollten nicht allgemein zugreifbar sein, um Änderungen an diesen Attributen, die möglicherweise die Invarianten der Klasse verletzen würden, zu unterbinden. Hätten wir sie aber als privat vereinbart, könnte auch die Klasse `ExtendedList` nicht auf die Attribute zugreifen. Dies ist aber notwendig, beispielsweise um die Methode `cut` zu implementieren.

Programmiersprachen, die Kapselung und Vererbung unterstützen, bieten deshalb in der Regel Konstrukte an, mit denen der Zugriffsbereich von Programmelementen auf die eigene Klasse und deren Subklassenhierarchie beschränkt werden kann. In Java gibt es die Möglichkeit, Programmelemente mit dem Zugriffsmodifikator `protected` als *geschützt* zu vereinbaren. Geschützte Programmelemente sind in Java in der eigenen Klasse, in allen direkten und indirekten Subklassen sowie im umfassenden Paket zugreifbar. *Geschützter Zugriff schließt also paketlokalen Zugriff mit ein.*

Grundsätzlich wird man in der objektorientierten Programmierung Klassen so entwerfen, dass sie für Vererbung offen und geeignet sind. Dementsprechend gewährt man den Softwareentwicklern, die Vererbung nutzen, um

*geschützt*

Subklassen zu realisieren, im Allgemeinen mehr Zugriff als normalen Benutzern von Klassen. Insbesondere wird man in Klassen, die für Vererbungsnutzung bestimmt sind, den Zugriff von Programmelementen in der Regel als geschützt und eher selten als privat vereinbaren.

**Unveränderliche Klassen und Methoden.** Das Überschreiben von Methoden ermöglicht es, das Verhalten eines Klassentyps völlig zu verändern, insbesondere auch außerhalb des Pakets, in dem der Klassentyp deklariert wurde. Betrachten wir das folgende Paket `soweitAllesOk`:

```
package soweitAllesOk;

public class A_nicht_Null {
    protected int a = 1;

    public int getA() { return a; }
    protected void setA( int i ) {
        if( i > 0 ) a = i;
    }
}

public class Anwendung {
    ...
    public static void m( A_nicht_Null ap ){
        float f = 7 / ap.getA();
    }
}
```

Bei der Klasse `A_nicht_Null` scheint man davon ausgehen zu können, dass das Attribut `a` immer echt größer 0 ist. D.h., dass die Invariante  $a > 0$  gilt. Denn `a` wird mit 1 initialisiert und nur gesetzt, wenn der neue Wert größer als 0 ist. (Im Übrigen darf ein normaler Nutzer auf die geschützte Methode `setA` nicht zugreifen.) Deshalb wird in der Anwendung darauf verzichtet zu prüfen, ob `ap.getA()` den Wert 0 liefert. Durch Vererbung lässt sich die Kapselung von `a` aber aufbrechen. Betrachten wir dazu folgendes Paket:

```
package einHackMitZweck;
import soweitAllesOk.*;

public class A_doch_Null extends A_nicht_Null {
    public int getA() { return -a; }
    public void setA( int i ) { a = i; }
}
```

```

public class Main {
    public static void main( String[] args ) {
        A_doch_Null adn = new A_doch_Null();
        adn.setA( 0 );
        A_nicht_Null ann = adn;
        Anwendung.m(ann);
    }
}

```

Der Aufruf von `Anwendung.m(ann)` in der letzten Zeile liefert eine arithmetische Ausnahme, ausgelöst von einer Division durch 0. Die Subklasse `A_doch_Null`, deren Objekte ja auch zum Typ `A_nicht_Null` gehören, hat die Methode `setA` durch Überschreiben verändert und außerdem noch öffentlich zugänglich gemacht. (Darüber hinaus wurde die Bedeutung der Methode `getA` völlig verändert.) Insgesamt zeigt das Beispiel, dass man in Subklassen Objekte definieren kann, die zwar zum Typ der Superklasse gehören, dessen Invariante aber nicht mehr erfüllen. Dies kann insbesondere dazu führen, dass die Methoden, die auf der Superklasse aufbauen, nicht mehr funktionieren.

Will ein Programmierer also sicher stellen, dass seine Klasse nicht durch Vererbung geändert werden kann, um z.B. zu gewährleisten, dass Invarianten nicht verletzt werden können und Verhalten nicht unerwünscht verändert werden kann (s. `getA`), braucht er ein zusätzliches sprachliches Hilfsmittel, denn dafür reicht es nicht einmal aus, das Schlüsselwort `protected` nicht zu verwenden (s. `getA`).

Java bietet hierfür das Schlüsselworts `final` an. Methoden und Klassen können durch Voranstellen des Schlüsselworts `final` als *unveränderlich* deklariert werden. Unveränderliche Methoden können nicht überschrieben werden. Von unveränderlichen Klassen können keine Subklassen gebildet werden. Typisches Beispiel für unveränderliche Klassen in der Java-Bibliothek ist die Klasse `String`.

*unveränderliche Klassen*

### 3.3.3.2 Zusammenspiel von Vererbung und Kapselung

Das Zusammenspiel von Vererbung und Kapselung ist komplexer, als man dies vielleicht auf den ersten Blick vermuten würde. Eine wichtige Frage bei diesem Zusammenspiel ist die folgende: Werden `private` Attribute und Methoden vererbt? Betrachten wir zunächst `private` Attribute: Sie sind in Subklassen nicht sichtbar und nicht zugreifbar. Insofern werden sie nicht vererbt. Andererseits müssen die Objekte der Subklasse auch die privaten Attribute der Superklasse besitzen, da sonst die ererbten Methoden nicht mehr korrekt arbeiten. Insofern werden `private` Attribute vererbt. Die übliche Java-Terminologie orientiert sich an der ersten Sichtweise. Wir vertreten hier die zweite Sichtweise, da wir Vererbung konzeptionell als automatische Über-

nahme von Programmteilen verstehen und – semantisch betrachtet – private Attribute in die Subklasse übernommen werden.

Selbstverständlich könnte man eine objektorientierte Sprache auch so entwerfen, dass Subklassen-Objekte die privaten Attribute nicht besitzen. Dann müsste man allerdings die Vererbung aller Methoden unterbinden, die auf die privaten Attribute zugreifen, und entsprechend die Aufrufe der Superklassen-Methoden und Konstruktoren mittels `super` verbieten. Abgesehen davon, dass man dadurch viel Potential für Vererbung aufgibt, bringt das zusätzliche Verpflichtungen für den Implementierer mit sich, da die Anforderungen des Subtypings nicht mehr automatisch erfüllt sind (vgl. S. 173).

Zur Motivierung der Sichtweise, dass private Attribute vererbt werden, private Methoden aber nicht, benutzen wir ein Programm, mit dem man Paare und Triple von `int`-Werten erzeugen sowie jeweils die Summe der Komponenten bilden kann. Die Triple sind dabei als Spezialisierung von Paaren realisiert:

```
class IntPair {
    private int a;
    private int b;
    IntPair( int ai, int bi ){ a = ai; b = bi; }
(*) int sum(){ return this.add(); }
    private int add(){ return a+b; }
}

class IntTriple extends IntPair {
    private int a;
    IntTriple( int ai,int bi,int ci ){ super(ai,bi); a = ci; }
    int sum(){ return this.add(); }
(+) private int add(){ return super.sum() + a; }
}

public class PrivateTest {
    public static void main( String[] arg ) {
        IntPair ip = new IntPair(3,9);
        IntTriple it = new IntTriple(1,2,27);
        System.out.println( ""+ip.sum()+" "+it.sum() );
    }
}
```

Jedes `IntTriple`-Objekt hat drei Attribute: die beiden Attribute `a` und `b` aus der Klasse `IntPair` und das zusätzlich deklarierte Attribut `a`.

Bei privaten Methoden erscheint es leichter, sich darauf zu verständigen, dass sie nicht vererbt werden. Gibt es in der Subklasse eine Methode, die den gleichen Namen wie eine private Methode der Superklasse hat, werden diese Methoden als unterschiedlich betrachtet, d.h. als wenn sie verschiedene



Namen hätten. Die Subklassen-Methode ist eine Neudefinition, die insbesondere die Superklassen-Methode nicht überschreibt. Dementsprechend findet zwischen diesen Methoden auch keine dynamische Methodenauswahl statt. Betrachten wir dazu den Aufruf `super.sum()` in der mit (+) markierten Zeile. Er zieht einen Aufruf der Methode `sum` in Zeile (\*) nach sich, also die Auswertung des Ausdrucks `this.add()`. Dabei referenziert der `this`-Parameter ein `IntTriple`-Objekt. Trotzdem wird die `add`-Methode der Klasse `IntPair` ausgeführt; denn sie wurde von keiner Methode überschrieben (insbesondere nicht von der `add`-Methode in der mit (+) markierten Zeile).

Anders sähe es aus, wenn beide `add`-Methoden nicht-privat wären. Dann würde im beschriebenen Fall in Zeile (\*) dynamisch gebunden, also die Methode von Zeile (+) ausgeführt – mit dem Ergebnis einer nicht terminierenden Rekursion und dem Überlauf des Laufzeitkellers. Insgesamt haben wir es also mit einem Beispiel zu tun, bei dem die Kapselung Einfluss auf die Ausführungssemantik des Programms hat.

Noch komplexer wird das Zusammenspiel von Vererbung und Kapselung in Java, wenn man die Vererbung von paketlokalen Methoden betrachtet, da diese in Subklassen innerhalb des eigenen Pakets sichtbar und zugreifbar sind, in Subklassen außerhalb des Pakets aber nicht. Wechselt eine Kette von Subklassen zwischen zwei Paketen hin und her, ergeben sich semantische Unklarheiten, die im Java-Sprachbericht nicht beseitigt sind und auch zu unterschiedlichen Java-Implementierungen geführt haben (vgl. [MPH98]).

### 3.3.3.3 Realisierung gekapselter Objektgeflechte

Im Folgenden werden wir Kapselungsprobleme betrachten, die speziell im Kontext von Objektgeflechten und auch darauf aufbauenden Komponenten (s.u.) auftreten. Dabei werden wir zunächst diskutieren, welche Probleme dabei trotz der bisher kennengelernten Möglichkeiten zur Kapselung im Sinne des Information Hiding (s. Abschn. 2.2.1, S. 111) entstehen können. Wir zeigen dann auf, welche Möglichkeiten zur Problembehebung greifen, wenn es keine Vererbung gäbe. Anschließend demonstrieren wir, wie Vererbung zu neuen Problemen führen kann und wie auch die gelöst werden können.

Ziel guter Programmierung ist es, Software in *Komponenten* mit klar festgelegten Schnittstellen zu strukturieren. In vielen Fällen wird die Funktionalität einer Software-Komponente, d.h. ihr Leistungsumfang, nicht von einem, sondern von mehreren Objekten erbracht. Einige dieser Objekte bilden die *Komponentenschnittstelle*; insbesondere gibt es von außerhalb der Komponente Referenzen auf die Objekte der Komponentenschnittstelle. Über die von außen zugreifbaren Methoden und Attribute dieser Objekte können andere Programmteile die Komponente benutzen.

Normalerweise besitzt eine Komponente auch Objekte, die hinter der Komponentenschnittstelle gekapselt und von außen nicht zugreifbar sind

bzw. sein sollten. Ein einfaches Beispiel einer gekapselten Komponente haben wir bereits mit der Klasse `LinkedList` kennen gelernt (vgl. Abb. 3.10, S. 221). Die Komponentenschnittstelle einer Instanz der Komponente wird vom `LinkedList`-Objekt und den `ListIterator`-Objekten gebildet; die `Entry`-Objekte sind gekapselt und von außen nicht zugreifbar. Abbildung 3.16 veranschaulicht diesen Sachverhalt für eine drei-elementige Liste; die Komponentenschnittstelle und damit der gekapselte Bereich sind durch eine Ellipse angedeutet<sup>19</sup>.

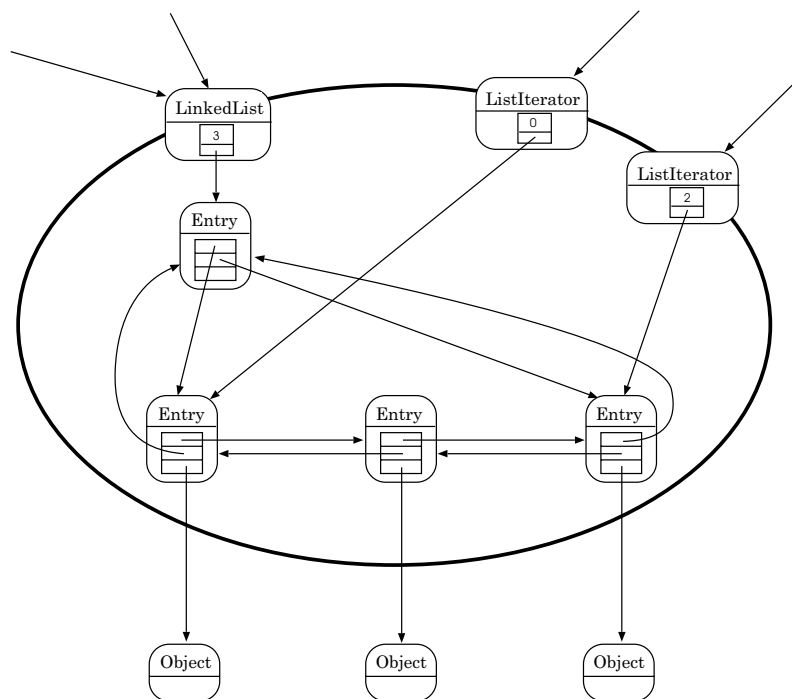


Abbildung 3.16: Gekapselte Liste mit Komponentenschnittstelle

**Kapseln der Repräsentation.** Diejenigen Teile einer Komponente, die nicht zur Schnittstelle gehören, bezeichnen wir als *Repräsentation* der Komponente. Die Repräsentation muss alle Instanzvariablen bzw. Objekte umfassen, von denen das korrekte Verhalten der Komponente abhängt. Sie kann darüber hinaus weitere Objekte enthalten; allerdings ist es im Allgemeinen von Vorteil, die Repräsentation auf die notwendigen Teile zu beschränken. Beispielsweise gehören zur Repräsentation einer doppelverketteten Liste deren `Entry`-Objekte sowie die geschützten Attribute der `LinkedList`- und `ListIterator`-Objekte. Wie in Abb. 3.16 angedeutet, betrachten wir die Objekte, die in der Liste eingetragen sind, nicht als Teil der Repräsentation, da

<sup>19</sup> Aus graphischen Gründen ist die Reihenfolge der Attribute von `Entry`-Objekten und des `LinkedList`-Objekts anders, als sie in Abb. 3.10 deklariert wurde.

die Funktionsfähigkeit der Listenimplementierung nicht vom Zustand dieser Objekte abhängt. D.h. weder die Verkettung der Listenelemente, noch der Inhalt der Attribute `header` und `size` der Klasse `LinkedList`, noch die Funktionsweise des Iterators sind abhängig von einer Änderung des Zustands der in dieser Liste gespeicherten Objekte. Objekte außerhalb der Komponente, wie die in einer Instanz von `LinkedList` gespeicherten Objekte, die von innerhalb der Repräsentation referenziert werden, nennt man im Englischen häufig „argument objects“. Die Funktionsfähigkeit einer Komponente darf nicht von derartigen Objekten abhängen (im Englischen verlangt man „no argument dependence“).

Das Kapseln der Repräsentation ist ein wichtiges Ziel guter Programmierung (im Englischen wird dieses Ziel auch mit „no representation exposure“ bezeichnet [NVP98]). Eine saubere Kapselung soll die fehlerhafte oder böswillige Benutzung einer Komponente verhindern. Außerdem ermöglicht eine wohl definierte Schnittstelle, dass Änderungen an der Repräsentation durchgeführt werden können, ohne dass Benutzer davon betroffen sind (vgl. Abschn. 2.2.1, S. 111). Daher muss eine saubere Kapselung insbesondere sicherstellen, dass es von außerhalb der Komponente keine direkten, unerwünschten Referenzen auf Objekte in der Repräsentation gibt, ein Problem, das wir bisher noch nicht explizit angesprochen haben. Man muss sich also davor schützen bzw. kontrollieren, dass außerhalb der Komponente kein Alias eines Repräsentationsobjekts vorhanden ist (vgl. Abschn. 2.1.2, S. 77); im Englischen spricht man deshalb in diesem Zusammenhang häufig von „alias protection“ bzw. „alias control“. Ein derartiger Alias auf ein Objekt der Repräsentation kann gefährlich sein, weil er es Objekten außerhalb der Komponente ermöglichen kann, die Repräsentation zu modifizieren, ohne eine Methode der Komponentenschnittstelle zu benutzen. Referenzen auf Objekte der Repräsentation nennen wir im Folgenden *schützenswerte* Referenzen.

Im Fall der Klasse `LinkedList` sind alle Repräsentationsobjekte durch unsere bisher kennengelernten Kapselungsmechanismen zunächst gegen direkten Zugriff auf Attributebene geschützt. Alle `Entry`-Objekte werden nur innerhalb der Klasse `LinkedList` erzeugt. Insbesondere kann kein `Entry`-Objekt von außerhalb von `LinkedList` durch einen Methodenaufruf in eine Instanz von `LinkedList` gelangen und es wird auch keine Referenz auf ein `Entry`-Objekt durch eine Methode nach außen gegeben. Daher kann es von außen keine Aliase auf `Entry`-Objekte geben.

Nicht immer können Aliase auf Repräsentationsobjekte so einfach vermieden werden, wie im Fall von `LinkedList`. Einen solchen Fall werden wir im nächsten Absatz kennen lernen zusammen mit den durch Aliasing hervorgerufenen Problemen und Ansätzen zu deren Lösung.

**Anwendung von Kapselungskonstrukten.** Für die Behandlung wichtiger Aspekte der Kapselung von Objektgeflechten betrachten wir als Beispiel ei-

ne einfache Auftragsverwaltung. Die Auftragsverwaltung nimmt von außen Aufträge entgegen und speichert sie in einer Liste. Sie zählt die eingegangenen Aufträge. Jeder Auftrag hat eine Beschreibung, eine Referenz auf den Kunden, der den Auftrag gegeben hat, und ein Attribut `zustand`, das darüber Auskunft gibt, ob der Auftrag auf die Bearbeitung wartet, ob er in Arbeit ist oder ob er erledigt wurde. Der Kunde soll jederzeit die Möglichkeit haben abzufragen, was der aktuelle Zustand seiner Aufträge ist. Über die Methode `naechstenAuftragHolen` kann sich eine Bearbeitungskomponente eine Referenz auf den nächsten wartenden Auftrag abholen. Dabei wird der Zustand des Auftrags auf „in Arbeit“ gesetzt. Außerdem zählt die Auftragsverwaltung, wieviele Aufträge zur Bearbeitung gegeben wurden. Die Bearbeitung setzt den Zustand auf „erledigt“, sobald die Bearbeitung beendet ist.

Abbildung 3.17 zeigt eine einfache Realisierung der skizzierten Aufgabenstellung. Dabei gehen wir davon aus, dass alle angegebenen Klassen im Paket `auftragsbehandlung` deklariert sind, dass die Bearbeitungskomponente (hier nicht gezeigt) ebenfalls zu diesem Paket gehört, dass aber die Auftraggeber in anderen, externen Paketen implementiert sind.

Die Repräsentation einer Auftragsverwaltung besteht aus deren Attributen, allen Objekten der Liste sowie den `Auftrag`-Objekten. Die von den Auftrag-Objekten referenzierten String- und `Personen`-Objekte brauchen wir in unserer Realisierung nicht zur Repräsentation zu rechnen, da das Verhalten der Auftragsverwaltung von ihnen nicht abhängt. Im Rahmen einer erweiterten Realisierung, in der die Kundeninformation relevant ist, müsste man ggf. anders entscheiden und die Person-Objekte mit einkapseln. Bei den String-Objekten ist es irrelevant, ob man sie zur Repräsentation rechnet oder nicht. Da der Zustand von String-Objekten nicht verändert werden kann, sind sie bereits bestmöglich geschützt (im Englischen nennt man solche Objekte „immutable“ oder „value objects“).

In einer Auftragsverwaltung wird man davon ausgehen, dass die Anzahl der eingegangenen Aufträge (`anzahlAuftraege`) größer oder gleich der Anzahl von den Aufträgen ist, die zur Bearbeitung abgeholt wurden (`anzahlBearbeitungen`). Bei der Analyse der Implementierung von Abb. 3.17 werden wir uns auf die Einhaltung dieser einfachen Invariante konzentrieren, also darauf, dass gilt:  $\text{anzahlAuftraege} \geq \text{anzahlBearbeitungen}$ .

Der kritische Punkt der Implementierung von Abb. 3.17 besteht darin, dass sowohl die Auftraggeber als auch die Bearbeitungskomponente Referenzen auf `Auftrag`-Objekte, also auf Objekte der Repräsentation erhalten. Bei den Auftraggebern geschieht dies dadurch, dass sie das `Auftrag`-Objekt selbst erzeugen. Dieses Objekt wird dann in die Repräsentation der Auftragsverwaltung eingebaut, also quasi von außen in die Kapsel importiert. Die Bearbeitungskomponente erhält die Referenz auf ein `Auftrag`-Objekt als Ergebnis einer Methode der Auftragsverwaltung, d.h. in diesem Fall wird eine

```
package auftragsbehandlung;

enum Status {
    WARTEND, INARBEIT, ERLEDIGT;
}

public class Auftrag {
    String beschreibung;
    Person kunde;
    Status zustand;

    public Auftrag( String b, Person k ) {
        beschreibung = b;
        kunde = k;
        zustand = Status.WARTEND;
    }
    public Status getZustand() { return zustand; }
    void setZustand( Status s ) { zustand = s; }
}

public class Auftragsverwaltung {
    protected LinkedList auftraege = new LinkedList();
    protected int anzahlAuftraege = 0;
    protected int anzahlBearbeitungen = 0;

    public void inAuftragGeben( Auftrag a ) {
        anzahlAuftraege++;
        auftraege.addLast( a );
    }
    Auftrag naechstenAuftragHolen() {
        LinkedList.ListIterator lit = auftraege.listIterator();
        while( lit.hasNext() ) {
            Auftrag atrg = (Auftrag) lit.next();
            if( atrg.getZustand() == Status.WARTEND ) {
                atrg.setZustand( Status.INARBEIT );
                anzahlBearbeitungen++;
                return atrg;
            }
        }
        return null;
    }
}
```

Abbildung 3.17: Einfache Auftragsverwaltung

schützenswerte Referenz herausgegeben. In beiden Fällen entsteht die Gefahr, dass über eine solche Referenz der Zustand eines Auftrags von außen in unzulässiger Weise modifiziert werden könnte. Beispielsweise könnte das `zustand`-Attribut eines bereits erledigten Auftrags wieder auf „wartend“ gesetzt werden. Dies würde zu einer nochmaligen Bearbeitung des Auftrags führen und könnte die obige Invariante verletzen.

Da die Invarianten die korrekten Zustände einer Komponente charakterisieren, bedeutet das Verletzen einer Invariante im Allgemeinen, dass sich die Komponente in Folge völlig unbestimmt verhält, dass sie insbesondere möglicherweise falsche Ergebnisse produziert. Im betrachteten Beispiel könnte man sich vorstellen, dass die Anzahl der Aufträge den Preis widerspiegelt, den die Kunden zu zahlen haben, und dass die Anzahl der Bearbeitungen die Kosten repräsentiert, die die Bearbeitungskomponenten in Rechnung stellen. Das Verletzen der Invariante bedeutet dann, dass die Verwaltung von den Kunden weniger erhält, als sie für die Bearbeitung zu zahlen hat.

Im Folgenden erläutern wir anhand der Implementierung aus Abb. 3.17, wie man diesen Gefahren mit den uns bereits bekannten Hilfsmitteln der Zugriffsbeschränkung und der zusätzlichen Anwendung von Codeinspektion begegnen kann.

Die Implementierung von Abb. 3.17 begegnet den oben beschriebenen Gefahren dadurch, dass sie für die Attribute der `Auftrag`-Objekte und für die Methode `setZustand` nur paketlokalen Zugriff zulässt. Da wir davon ausgegangen sind, dass Auftraggeber außerhalb des Pakets `auftragsbehandlung` realisiert sind, haben sie keinen Zugriff auf die Attribute von `Auftrag`-Objekten und können auch die Methode `setZustand` nicht aufrufen. Dieser Schutz ist vor allem auch notwendig, weil die Implementierung der Auftraggeber im Allgemeinen unbekannt ist. Anders verhält es sich bei der Bearbeitungskomponente, die wir als Teil des Pakets `auftragsbehandlung` angenommen haben. Ihr Programmcode ist allerdings bekannt, sodass geprüft werden kann, ob sie die Methode `setZustand` korrekt benutzt (Codeinspektion).

Die Zugriffseinschränkungen der Klasse `Auftrag` und die Prüfung der Verwendung von `setZustand` in der Bearbeitungskomponente sind sicherlich wichtige Voraussetzungen, um ein korrektes Verhalten der Auftragsverwaltung zu erreichen. Wie wir im folgenden Absatz sehen werden, reichen sie aber nicht aus, wenn zusätzlich Vererbung ins Spiel kommt.

**Durch Vererbung hervorgerufene Problemfälle mit schützenswerten Referenzen.** Das Importieren von Objekten in die Repräsentation und das Herausgeben von schützenswerten Referenzen sind kritische Operationen. Typische Probleme entstehen im Zusammenhang mit Vererbung und dynamischer Bindung.

**Problem durch Subklassen von Auftrag.** Betrachten wir einen böswilligen Auftraggeber, der erreichen will, dass seine Aufträge doppelt bearbeitet werden, auch wenn sie nur einmal gestellt wurden. Anstatt Auftrag-Objekten könnte er Objekte der folgenden Subklasse übergeben:

```
public class SubAuftrag extends Auftrag {
    boolean einmalGetaeuscht = false;

    public SubAuftrag( String b, Person k ) { super( b, k); }

    public Status getStatus() {
        Status wahrerZustand = super.getStatus();
        if( wahrerZustand==Status.ERLEDIGT
            && !einmalGetaeuscht ) {
            einmalGetaeuscht = true;
            return Status.WARTEND;
        } else {
            return wahrerZustand;
        }
    }
}
```

Die Auftragsverwaltung merkt nicht, dass sie Objekte der Klasse `SubAuftrag` übergeben bekommt. Da sie die Methode `getStatus` benutzt, um den Zustand abzufragen, bei `SubAuftrag`-Objekten aber die obige überschreibende Implementierung verwendet wird, wird ihr einmal vorgespiegelt, ein bereits erledigter Auftrag sei noch wartend. Das Problem hier besteht darin, dass sich importierte Objekte nicht unbedingt so verhalten müssen, wie man es erwartet.

Zur Lösung dieses Problems kann man die Auftragsklasse intern in der Auftragsverwaltung realisieren z.B. durch eine private innere Klasse `AuftragIntern`, wie in Abb. 3.18 gezeigt. Ein Auftragsobjekt wird dann nicht mehr vom Auftraggeber erzeugt und in die Kapsel importiert. Statt dessen wird von der Auftragsverwaltung selbst ein passendes Auftragsobjekt erzeugt. Die vom Auftraggeber benötigten Angaben zur Auftragsbeschreibung und zu seiner Person werden der Auftragsverwaltung mittels der Methode `inAuftragGeben` einzeln als Parameter übergeben. Da der Auftraggeber jederzeit in der Lage sein soll, den Zustand seines Auftrags abzufragen, bekommt er von der Methode `inAuftragGeben` eine Referenz auf das Auftragsobjekt zurück. Allerdings erlaubt ihm diese Referenz vom (geänderten) Typ `Auftrag` nur, den Zustand des Auftrags auszulesen:

```
public interface Auftrag {
    Status getStatus();
}
```

**Problem durch Subklassen der Bearbeitungskomponente.** Auch das Herausgeben von schützenswerten Referenzen birgt etliche Fehlerquellen. Es reicht nicht zu prüfen, ob die Methode `setZustand` korrekt verwendet wird. Vielmehr muss auch genau kontrolliert werden, wer Zugriff auf die schützenswerten Referenzen bekommt und ob zukünftige Subklassen der Bearbeitungskomponente, also nicht einsehbarer Programmcode, mittels dieser Referenzen die `Auftrag`-Objekte und damit die Repräsentation der Auftragsverwaltung in unzulässiger Weise manipulieren können.

Um dieser Gefahr zu begegnen, bekommt die Bearbeitungskomponente keine direkte Referenz auf ein Auftragsobjekt mehr, über das sie die Methode `setZustand` aufrufen kann. Statt dessen wird der Zugriff von Bearbeitungskomponenten auf Auftragsobjekte über sogenannte Auftragswrapper-Objekte<sup>20</sup> (vgl. Abb. 3.18) geregelt, die nur einen eingeschränkten Zugriff erlauben. Ein Auftragswrapper besitzt eine Referenz auf das zugehörige `AuftragIntern`-Objekt und erlaubt es, dessen Beschreibung auszulesen und zu melden, dass die Bearbeitung erledigt ist (`erledigtMelden`). Die Änderung des `zustand`-Attributes wird von außen durch Aufruf von `erledigtMelden` nur veranlasst; der Aufruf von `setZustand`, der die Änderung bewirkt, geschieht innerhalb der Kapsel und ist somit unter vollständiger Kontrolle der Verwaltungskomponente. Dies kann im Übrigen dadurch erzwungen werden, dass `setZustand` als `private` Methode deklariert wird.

**Die Lösung beider Probleme zusammengefasst.** Der Programmcode aus Abb. 3.18 zeigt eine Lösung für beide der hier angesprochenen Probleme.

Wie das Beispiel andeutet, bietet die Wrapper-Technik mehr Flexibilität als die Benutzung einschränkender Schnittstellen. Wrapper können eine recht umfangreiche Funktionalität anbieten, die sogar auf diejenigen Objekte zugeschnitten werden kann, an die eine Referenz auf die Wrapper übergeben wird. Zum Beispiel könnte man die Auftragswrapper mit einem Mechanismus ausstatten, der die „erledigt“-Meldung nur von derjenigen Bearbeitungskomponente akzeptiert, an die das Wrapperobjekt herausgegeben wurde.

---

<sup>20</sup>Im Gegensatz zu den Wrapper-Klassen in `java.lang` (vgl. S. 172) geht es hier also nicht darum, Werte in Objekte einzubetten, sondern den Zugriff auf ein anderes Objekt „einzuwickeln“; manchmal nennt man ein solches Objekt auch einen Proxy.



```

package auftragsbehandlungV2;

public interface Auftrag {
    Status getStatus();
}

public class Auftragsverwaltung {
    protected LinkedList auftraege = new LinkedList();
    protected int anzahlAuftraege = 0;
    protected int anzahlBearbeitungen = 0;

    private class AuftragIntern implements Auftrag {
        private String beschreibung;
        private Person kunde;
        private Status zustand;

        private AuftragIntern( String b, Person k ) {
            beschreibung = b;
            kunde = k;
            zustand = Status.WARTEND;
        }

        public Status getStatus() { return zustand; }
        private void setStatus( Status s ) { zustand = s; }
    } // END class AuftragIntern

    class Auftragswrapper {
        private AuftragIntern atrg;
        private Auftragswrapper( AuftragIntern a ) { atrg = a; }
        String getBeschreibung() { return atrg.beschreibung; }
        void erledigtMelden() {
            atrg.setStatus( Status.ERLEDIGT );
        }
    }

    public Auftrag inAuftragGeben( String b, Person k ) {
        AuftragIntern a = new AuftragIntern(b,k);
        anzahlAuftraege++;
        auftraege.addLast( a );
        return a;
    }

    Auftragswrapper naechstenAuftragHolen() {
        LinkedList.ListIterator lit = auftraege.listIterator();
        while( lit.hasNext() ) {
            AuftragIntern atrg = (AuftragIntern) lit.next();
            if( atrg.getStatus() == Status.WARTEND ) {
                atrg.setStatus( Status.INARBEIT );
                anzahlBearbeitungen++;
                return new Auftragswrapper( atrg );
            }
        }
        return null;
    }
}

```

Abbildung 3.18: Gekapselte Auftragsverwaltung

**Fazit zur Realisierung gekapselter Komponenten.** Bei der Realisierung gekapselter Komponenten sollte man immer folgende Grundregeln beachten:

1. Der Zugriff auf die Attribute der Repräsentation, die zu Objekten der Komponentenschnittstelle gehören, muss Benutzern verwehrt sein. Beispielsweise sind die Attribute der `LinkedList`- und `ListIterator`-Objekte als privat bzw. geschützt deklariert.
2. Soweit möglich, sollten Objekte, die außerhalb der Komponente erzeugt wurden, nicht in die Repräsentation eingebaut werden.
3. Soweit möglich, sollten Referenzen auf Objekte der Repräsentation nicht an Programmteile außerhalb der Komponente herausgegeben werden.

In sehr vielen Fällen lässt es sich vermeiden, dass Programmteile außerhalb einer gekapselten Komponente Zugriff auf deren schützenswerte Referenzen erhalten. Objekte, die der Komponente als Parameter übergeben werden, können beispielsweise nur als Kopie in die Repräsentation eingebaut werden. Anstatt direkt Objektreferenzen herauszugeben, kann man vielfach auch die entsprechende Information geeignet codieren. Beispielsweise könnte die Auftragsverwaltung den Aufträgen eindeutige Nummern geben, intern eine Abbildung von Auftragsnummern auf Objekte realisieren und nach außen nur über Auftragsnummern kommunizieren.

In manchen Fällen möchte man aber auch bei gekapselten Objekten die Bequemlichkeit oder Effizienz nicht missen, die man durch Objektreferenzen erreicht. Dann muss man durch Anwendung der Kapselungs- und Programmierkonstrukte objektorientierter Sprachen den Zugriff über solche schützenswerten Referenzen geeignet einschränken. Wir haben dazu im Rahmen der Auftragsverwaltung zwei gängige Programmiertechniken kennen gelernt, mit denen man die skizzierten Probleme der Auftragsverwaltung lösen kann. Die eine Technik besteht darin, die gekapselten Objekte mittels einer außerhalb der Komponente nicht sichtbaren Klasse zu implementieren und nach außen nur eine eingeschränkte Schnittstelle auf diese Objekte zur Verfügung zu stellen. Diese Technik wurde für die Auftragsobjekte verwendet und zwar in Bezug auf die Schnittstelle zum Auftraggeber. Die andere Technik besteht darin, die Komponentenschnittstelle zu erweitern, indem man eine zusätzliche Schicht zwischen den gekapselten Objekten und den außerhalb der Komponente liegenden Objekten einführt und zwar durch *Wrapper-Objekte*. Dadurch kann man den Zugriff auf Objekte der Repräsentation gezielt einschränken. Diese Technik wurde angewendet, um eine passende, gefahrlose Schnittstelle zur Bearbeitungskomponente zu schaffen.

**Zusammenhang Kapselung, Vererbung und Wiederverwendung.** Vererbung und Subtyping einerseits und Kapselung andererseits stehen in einem Spannungsverhältnis. Macht man bei der Realisierung der Kapselung einer Komponente viel Gebrauch vom privaten Zugriffsmodus, schränkt man damit die Spezialisierungsmöglichkeiten in Subklassen erheblich ein. Gibt man andererseits den Subklassen zuviel Freiheiten und klärt die Entwickler nicht darüber auf, welche Bedingungen und Invarianten sie einzuhalten haben, kann ein Aufweichen der Kapselung oder ein fehlerhaftes Verhalten nur mit viel Glück vermieden werden.

Um das angesprochene Spannungsverhältnis an einem kleinen Beispiel zu illustrieren, betrachten wir das geschützte Attribut `auftraege` der Auftragsverwaltung. Man kann sich unschwer Spezialisierungen vorstellen, die auf dieses Attribut zugreifen müssen, z.B. verfeinerte Auftragsverwaltungen, die komplexere Aufträge zunächst in mehrere Teilaufträge zerlegen. Wäre das Attribut privat, ließen sich derartige Spezialisierungen nicht ohne weiteres mittels Vererbung realisieren. Haben Subklassen andererseits Zugriff auf die Auftragsliste, können sie diese schützenswerte Referenz als Ergebnis einer zusätzlichen Methode `getAuftraege` unbedacht nach außen geben und damit die Kapselung aufbrechen.

Ein anderes Spannungsverhältnis existiert zwischen Kapselung und Wiederverwendung. Am besten lassen sich Programmteile kapseln, die man selbst implementiert hat, da man dann deren Zugriffsrechte festlegen kann. Bei Verwendung von Bibliotheksklassen oder Wiederverwendung anderer Klassen muss man normalerweise davon ausgehen, dass diese Klassen öffentlich und damit von allen zugreif- und manipulierbar sind. Zur Illustration betrachten wir wiederum die Auftragsliste. Da für ihre Implementierung die Klasse `LinkedList` (wieder-)verwendet wurde, kann jede Subklasse der Klasse `Auftragsverwaltung` eine Referenz auf die Auftragsliste herausgeben; der Typ dieser Referenz ist ja öffentlich bekannt. Wäre andererseits die Auftragsliste durch eine geschützte innere Klasse von `Auftragsverwaltung` implementiert worden, hätte die zusätzliche Subklassenmethode `getAuftraege` einen öffentlich nicht bekannten Ergebnistyp, was ihre Anwendbarkeit außerhalb der Auftragsverwaltung zumindest einschränken würde.

Wir werden auf den Zusammenhang zwischen Vererbung, Kapselung und Wiederverwendbarkeit objektorientierter Programmteile noch einmal in Abschnitt 3.4 zurückkommen. Zuvor greifen wir in den Abschnitten 3.3.4 und 3.3.5 noch einmal das Überschreiben und Überladen von Methoden auf. In Abschnitt 3.3.4 stellen wir das Überschreiben von Instanzmethoden dem sog. Verstecken von Attributen und Klassenmethoden gegenüber; in Abschnitt 3.3.5 wenden wir uns noch einmal dem Auflösen von Methodenaufrufen zu, insbesondere im Zusammenhang mit dem Überladen von Methoden.

### 3.3.4 Verstecken von Attributen und Klassenmethoden vs. Überschreiben von Instanzmethoden

Ähnlich wie für Instanzmethoden erlaubt Java auch für Attribute und Klassenmethoden die Deklaration namensgleicher Attribute und Klassenmethoden in Subtypen.<sup>21</sup> Anders als bei Instanzmethoden spricht man jedoch nicht vom Überschreiben, sondern vom *Verstecken* (engl. *hiding*) von Attributen und Klassenmethoden.<sup>22</sup> Tatsächlich besteht zwischen dem Überschreiben von Instanzmethoden und dem Verstecken von Attributen und Klassenmethoden ein wesentlicher Unterschied beim Zugriff. Führt man eine (möglicherweise überschriebene) Instanzmethode auf einem Objekt aus, hängt es vom dynamischen Typ des Objekts ab, welche Methode zur Ausführung kommt. Führt man eine (möglicherweise versteckte) Klassenmethode aus oder greift auf ein (möglicherweise verstecktes) Attribut zu, hängt es vom deklarierten Typ des Objekts ab, welche Methode zur Ausführung kommt bzw. auf welches Attribut der Zugriff erfolgt. Das folgende Beispiel, das sich an [Fla99] und [AG05] orientiert, illustriert diesen wichtigen Unterschied:

*Verstecken*

```
import java.io.*;

class Superclass {
    int i = 42;
    int f() { return i; }
    static String g() { return "Superclass"; }
}

class Subclass extends Superclass {
    int i = 4711;
    int f() { return -i; }
    static String g() { return "Subclass"; }
}

class Test {
    public static void main (String args[]) {
        Subclass sub = new Subclass();
        Superclass sup = sub;

        System.out.println(sub.i);
        System.out.println(sub.f());
        System.out.println(sub.g());
    }
}
```

<sup>21</sup>Im Falle von Methoden namens- und signaturgleicher Methoden.

<sup>22</sup>In der ersten Auflage der Java Sprachbeschreibung [GJS96] wird synonym zu „verstecken“ auch der Begriff „verschatten“ (shadowing) gebraucht. Die zweite Auflage [GJSB00] verwendet nur noch den Begriff „verstecken“ für das in diesem Abschnitt behandelte Phänomen.



```

class Test {
    public static void main (String args[]) {
        Subclass    sub = new Subclass();
        Superclass  sup = sub;

        System.out.println(sub.i);           // Referenziert Subclass.i;
                                              // druckt also 4711
        System.out.println(sub.f());          // Referenziert Subclass.f;
                                              // druckt also -4711
        System.out.println(sub.g());          // Referenziert Subclass.g;
                                              // druckt also Subclass

        System.out.println(sup.i);           // Referenziert Superclass.i;
                                              // druckt also 42
        System.out.println(sup.f());          // Referenziert Subclass.f;
                                              // druckt also -4711
        System.out.println(sup.g());          // Referenziert Superclass.g;
                                              // druckt also Superclass
    }
}

```

Wir beschließen dieses Beispiel mit dem Hinweis, dass die Aufrufe `sub.g()` und `sup.g()` in der Methode `main` der Klasse `Test` keinem guten Programmierstil folgen. Obwohl syntaktisch zulässig, entspricht es gutem Programmierstil, Klassenmethoden stets den Klassennamen als Präfix voranzustellen. Die Aufrufe `sub.g()` und `sup.g()` sollten daher durch die bedeutungsgleichen Aufrufe `Subclass.g()` und `Superclass.g()` ersetzt werden.

### 3.3.5 Auflösen von Methodenaufrufen im Kontext überschriebener und überladener Methoden

In Abschnitt 2.1.4 haben wir festgehalten, dass sich überladene Methoden in ihrer Signatur unterscheiden müssen. Auf diese Weise sei es möglich, die Überladung statisch, also zur Übersetzungszeit aufzulösen und für jede Aufrufstelle zu ermitteln, welche der namensgleichen Methoden an der Aufrufstelle tatsächlich gemeint ist. In diesem Abschnitt wollen wir das Auflösen überladener Methodenaufrufe noch einmal genauer untersuchen, insbesondere im Kontext von Vererbungshierarchien.

Generell ist ein Methodenaufruf einfach aufzulösen, wenn es genau eine an der Aufrufstelle sichtbare Methode mit passendem Namen und passender Parameterdeklaration gibt. Im Grundsatz gilt dies auch, wenn an der Aufrufstelle mehrere Methoden des passenden Namens sichtbar sind, diese sich aber jeweils in ihrer Signatur unterscheiden, also in der Situation überladener

Methodenaufrufe. Delikater wird das Problem jedoch im Zusammenhang mit Vererbung. Dazu betrachten wir folgendes Programmfragment als Beispiel:

```
class A { ... }

class B extends A { ... }
class C extends B { ... }

class D extends A { ... }
class E extends D { ... }

class Overload {
    void foo (A a, D d) { ... }    // Deklaration #1
    void foo (B b, A a) { ... }    // Deklaration #2
    void foo (C c, D d) { ... }    // Deklaration #3

    void overloadTest() {
        A a = new A();
        B b = new B();
        C c = new C();
        D d = new D();
        E e = new E();

        foo (a,d);    // Aufruf #1
        foo (c,a);    // Aufruf #2
        foo (c,e);    // Aufruf #3
        foo (b,d);    // Aufruf #4
    }
}
```

In diesem Beispiel ist der Name der Methode `foo` in der Klasse `Overload` überladen. Zusätzlich stehen die Typen der Methodenparameter in einer Vererbungsbeziehung. Intuitiv erwarten wir, dass der mit #1 bezeichnete Aufruf zu einem Aufruf der mit #1 deklarierten Methode `foo` führt, weil die Typen der Parameter an der Aufrufstelle exakt mit den Typen der Parameter der Deklaration dieser Methode übereinstimmen. Was ist aber, wenn wir den Aufruf #3 betrachten? Offenbar passen die Typen der Argumentparameter zu keiner der drei Deklarationen von `foo` exakt. Andererseits gilt, dass ein Objekt vom Typ `C` auch ein Objekt der Typen `B` und `A` ist, und ein Objekt vom Typ `E` auch ein Objekt der Typen `D` und `A`. Mithin ist jede der drei Deklarationen der Methode `foo` grundsätzlich passend, d.h. im Sinne der ist-ein-Relation der aktuellen und formalen Methodenparameter.

Um überladene Methodenaufrufe auflösen zu können, reicht es also i.a. nicht, dass der Übersetzer lediglich die verschiedenen Signaturen untersucht, er muss auch die (deklarierten) Typen der formalen und Argumentparameter in Betracht ziehen. Der Übersetzer folgt dazu einem sog. *most-specific*

Ansatz, d.h. im Sinne der Vererbungshierarchie wird jeweils die „speziellste“, die am besten passende Methode ausgewählt, so sie existiert. Andernfalls wird die Übersetzung mit einem Fehler abgebrochen. Im obigen Beispiel liefert dieses Verfahren die Methode #3, d.h. der dritte Aufruf von `f○○` führt zum Aufruf der als #3 deklarierten Methode `f○○`.

Im Folgenden beschreiben wir die Auflösung überladener Methodenaufrufe etwas genauer. Sie wird in zwei Schritten vorgenommen.

**Erster Schritt** Im ersten Schritt werden alle sichtbaren Methoden mit passendem Namen und passender Signatur bestimmt. Für die Signatur heißt das, dass die Typen der formalen Parameter (mittelbare) Supertypen der deklarierten Typen der Argumente oder - im Falle von Zahltypen - Zahltypen mit mindestens so großem Wertevorrat sein müssen. Wird in diesem Schritt genau eine solche Methode gefunden, dann wird diese aufgerufen; wenn keine solche Methode gefunden wird, wird die Übersetzung mit einem Fehler abgebrochen. Ansonsten wird mit dem zweiten Schritt fortgefahren, dem eigentlichen most-specific-Algorithmus.

**Zweiter Schritt** Der most-specific-Algorithmus streicht aus der im ersten Schritt bestimmten Menge nach und nach solche Methoden, für die es noch eine „speziellere“ Methode in der Menge gibt, d.h. eine Methode, deren Parametertypen (mittelbare) Subtypen bzw. Zahltypen mit gleichem oder kleinerem Wertevorrat sind. Bleibt nach diesem zweiten Schritt genau eine Methode übrig, so wird diese Methode aufgerufen. Ansonsten ist der Aufruf mehrdeutig, d.h. es existiert keine im Sinne der Vererbungshierarchie speziellste Methodendeklaration. In diesem Fall kann der Aufruf zur Übersetzungszeit nicht aufgelöst werden und die Übersetzung bricht mit einem Fehler ab.

Wendet man dieses Verfahren auf die vier Methodenaufrufe im obigen Beispiel an, so sind die Aufrufe #1 und #2 nach dem ersten Schritt des obigen Verfahrens erfolgreich aufgelöst, und Aufruf #3 nach dem zweiten Schritt. Der letzte Aufruf hingegen, Aufruf #4, kann nicht erfolgreich aufgelöst werden. Im ersten Schritt werden die beiden ersten Deklarationen von `f○○` als mögliche Kandidaten identifiziert. Keine dieser Deklarationen ist jedoch „spezieller“ als die jeweils andere. Folglich kann keine dieser Deklarationen im zweiten Schritt des most-specific-Verfahrens ausgeschieden werden. Der Aufruf bleibt mehrdeutig, die Übersetzung bricht mit einem entsprechenden Fehler ab.

Wir haben in diesem Abschnitt nicht alle Aspekte bei der Auflösung von Methodenaufrufen angesprochen, sondern uns bewusst auf die Übersetzungszeitaktivitäten konzentriert. Werden überladene Methoden auf Objekten aufgerufen, kann die tatsächlich ausgeführte Methode erst zur Laufzeit in Abhängigkeit des dynamischen Typs des Empfängerobjekts bestimmt werden. Auch in diesen Fällen ist aber schon zur Übersetzungszeit die Signatur



überladener Methodenaufrufe im Sinne des most-specific-Ansatzes anhand der deklarierten Typen aufgelöst worden. Liegt also der dynamische Typ des Empfängerobjekts zur Laufzeit offen, liegt auch die auszuführende überladene Methode fest. In diesem Sinne wird auch im Kontext dynamischer Methodenbindung Überladung statisch aufgelöst. I.a. zerfällt also das Auflösen von Methodenaufrufen in objektorientierten Sprachen in Übersetzungszeit- und in Laufzeitaktivitäten. Es ist hilfreich, sich diese Aktivitäten anhand einiger Beispiele, auch eigener, zu verdeutlichen. Überzeugen Sie sich dabei auch davon, dass der oben beschriebene most-specific-Ansatz weder Rückgabetyphen von Methoden noch geworfene Ausnahmen heranzieht, um überladene Methodenaufrufe zur Übersetzungszeit aufzulösen.

### 3.4 Objektorientierte Programmierung und Wiederverwendung

Als wichtiger Vorteil der objektorientierten Programmierung wird immer wieder die bessere Wiederverwendbarkeit objektorientierter Programm-Module genannt. Wiederverwendung kommt in zwei Varianten vor:

1. Bibliotheksvariante: Programm-Module werden von vornherein für Bibliotheken oder Software-Baukästen entwickelt, d.h. der Einsatz innerhalb anderer Programmsysteme ist der Regelfall (beispielsweise Datentypbibliotheken, Oberflächenbaukästen).
2. Adaptionsvariante: Ein Programm-Modul wurde ursprünglich für ein Programmsystem entwickelt und ist nun an veränderte Rahmenbedingungen anzupassen (wie in den Beispielen von Abschn. 3.3.1).

Drei Aspekte objektorientierter Programmiersprachen spielen im Zusammenhang mit Wiederverwendung eine Rolle.

1. Die Strukturierung der Typen durch Subtyping vereinfacht das Hinzufügen neuer Typen mit erweiterten Schnittstellen. In prozeduralen Sprachen führt das Hinzufügen neuer Typen häufig zu Änderungen am existierenden Programm (insbesondere sind oft Variantenverbunde anzupassen und Fallunterscheidungen zu ergänzen, die über Typinformation geführt werden), während im entsprechenden objektorientierten Programm keinerlei Änderungen nötig sind.
2. Vererbung bringt eine erhebliche Vereinfachung bei der Spezialisierung existierender Typen.

3. Kapselungsmechanismen, wie sie von vielen objektorientierten Sprachen unterstützt werden, verringern die Gefahr unzulässiger Zugriffe auf Datenkomponenten, was insbesondere wichtig ist, wenn Programm-Module in neuen Kontexten verwendet werden.

Objektorientierte Programmierung und objektorientierte Sprachen können also durchaus zur besseren Wiederverwendung von Programmen beitragen. Sie sind aber sicherlich kein Wundermittel. Gerade im Zusammenhang mit der Bibliotheksvariante spielen z.B. Parametrisierungs- und Modularisierungskonzepte eine sehr wichtige Rolle, und diese werden von objektorientierten Sprachen nicht besser unterstützt als von ihren prozeduralen Konkurrenten. Auch wird die Komplexität mächtiger Sprachkonstrukte, wie zum Beispiel von Vererbung, oft unterschätzt. Dies kann dazu führen, dass diese Sprachkonstrukte ungeschickt oder fehlerhaft angewandt werden, dass Werkzeuge (z.B. Compiler) derartige Sprachkonstrukte unterschiedlich oder falsch unterstützen und dass die maschinelle Analyse und Optimierung derartiger Programme erschwert wird.



# Kapitel 4

## Bausteine für objektorientierte Programme

Dieses und das folgende Kapitel beschäftigen sich mit der Anwendung objektorientierter Programmierkonstrukte zur Realisierung wiederverwendbarer Programmteile. In diesem Kapitel konzentrieren wir uns dabei auf Programmbausteine, die als einzelne Klassen, d.h. weitgehend unabhängig von anderen Bausteinen, verwendet werden können. Kapitel 5 behandelt dann das Zusammenwirken wiederverwendbarer Programmteile im Rahmen von Programmgerüsten.

Dieses Kapitel beginnt im ersten Abschnitt mit einer kurzen Einführung zum Thema „Programmbausteine“ und bietet in dem Zusammenhang eine Übersicht über die Java-Bibliothek. Der zweite Abschnitt erläutert dann anhand der Ausnahmebehandlung und der dazu verwendeten Objekte ein einfaches Beispiel für wiederverwendbare Bausteine. Der dritte Abschnitt behandelt Javas Stromklassen. Er liefert damit außer realistischem Anschauungsmaterial auch wichtige programmiertechnische Fähigkeiten.

### 4.1 Bausteine und Bibliotheken

Dieser Abschnitt bietet eine kurze Einführung zum Thema „Programmbausteine“ und eine Übersicht über die Java-Bibliothek.

#### 4.1.1 Bausteine in der Programmierung

Der Begriff „Programmbaustein“ wird selten mit einer klaren Bedeutung verwendet. Ihm zugrunde liegt die Vorstellung, Programme aus vorgefertigten Teilen zusammenzusetzen, ähnlich wie man es von Baukästen her oder in der industriellen Produktion kennt. Bei der Übertragung dieser Vorstellung in den Bereich der Programmkonstruktion spielen folgende Aspekte eine wichtige Rolle:

- Wie allgemein bzw. anpassbar sind die vorgefertigten Bauteile?
- Sind die Bauteile direkt einsatzbereit oder realisieren sie nur Teilaspekte und müssen vor der Anwendung komplettiert werden?
- Sind die Bauteile unabhängig voneinander, hierarchisch strukturiert oder sind sie wechselseitig aufeinander angewiesen?
- Wie werden die Bauteile zusammengesetzt?
- Wie sind die Bauteile beschrieben und welche Möglichkeiten gibt es, nach ihnen zu suchen?

Sehr allgemeine, relativ unabhängige Bausteine gibt es vor allem für den Bereich der Standarddatentypen wie Stapel, Schlangen, Listen, Mengen, Hash-Tabellen, Graphen, etc. In vielen Fällen können diese Bausteine so verwendet werden, wie man sie vorfindet. Demgegenüber stehen Bauteile, die relativ spezielle Aufgaben lösen und auf eine bestimmte Zusammenarbeit mit anderen Bauteilen angewiesen sind, etwa wie die Baugruppen innerhalb einer Autoserie (Fahrwerk, Motor, Karosserie). Im Softwarebereich denke man dabei beispielsweise an standardisierte Bausteine, um die Buchführung und Personalverwaltung von Unternehmen zu automatisieren.

**Beziehungen zwischen Bausteinen.** Bausteine können in sehr unterschiedlicher Beziehung zueinander stehen. Um die Diskussion zu konkretisieren, gehen wir in diesem Absatz von der vereinfachenden<sup>1</sup> Annahme aus, dass ein Baustein einer Klasse oder Schnittstelle entspricht. Wir nennen einen Baustein *unabhängig*, wenn in seiner öffentlichen und paketlokalen Schnittstelle nur Typen verwendet werden, die entweder vom Baustein selbst deklariert sind oder vordefinierte Typen der zugrunde liegenden Sprache sind (als vordefiniert betrachten wir in Java die Basisdatentypen und die Typen des Pakets `java.lang`). Beispielsweise stellt die Klasse `LinkedList` von Abb. 3.10, S. 221, einen unabhängigen Baustein dar. Unabhängige Bausteine kann man benutzen, ohne Zugriff auf weitere Typen zu haben.

Prinzipiell erleichtert Unabhängigkeit das Verständnis, die Anwend- und Wartbarkeit von Bausteinen. Trotzdem sind unabhängige Bausteine in objektorientierten Bibliotheken eher selten. Der Grund dafür liegt darin, dass man oft ähnlich geartete Bausteine benötigt, die man aber so weit möglich unter einer abstrakteren Schnittstelle subsumieren möchte. Konsequenterweise versucht man deshalb, ähnlich geartete Bausteine mittels Subtyp- bzw. Klassenhierarchien zu strukturieren. Wie wir am Beispiel der Klassen für die Ausnahmebehandlung sehen werden, führt dies zwar nicht notwendig zu Abhängigkeiten zwischen den Bausteinen in ihrer Schnittstellendeklaration, wie es z.B.

<sup>1</sup>In anderem Zusammenhang kann es durchaus sinnvoller sein, Bausteine mit Paketen oder einzelnen Objekten zu identifizieren.

*unabhängig*

bei Stromklassen der Fall ist (s.u.). Allerdings ist man beim Entwurf der Bausteine einer solchen Hierarchie in der Regel bemüht, die Parametertypen der Methoden so abstrakt wie möglich zu wählen, um eine allgemeine Verwendbarkeit zu erreichen. Wie wir am Beispiel der Stromklassen sehen werden, werden die spezielleren Bausteine in ihrer Schnittstellendeklaration dadurch von den abstrakteren Bausteinen in der Hierarchie abhängig. Bausteine, die in ihrer öffentlichen und paketlokalen Schnittstelle nur ihre Supertypen, selbst deklarierte oder vordefinierte Typen benutzen, nennen wir *eigenständig*.

*eigenständig*

Zur Lösung komplexerer softwaretechnischer Aufgabenstellungen reichen einzelne unabhängige oder eigenständige Bausteine häufig nicht aus. Benötigt werden dann gleichzeitig mehrere Bausteine, die eng zusammenarbeiten. Diese enge Kooperation schlägt sich syntaktisch in Form von komplexeren Abhängigkeiten nieder, beispielsweise verschränkt rekursiver Abhängigkeiten der Typen und Methoden. Auf solche *eng kooperierenden* Bausteine, die zusammen sogenannte *Programngerüste* bilden, werden wir in Kap. 5 näher eingehen.

*eng*

*kooperierend*

**Komposition von Bausteinen.** Identifiziert man Bausteine mit Klassen, bedeutet Komposition von Bausteinen im Wesentlichen, dass eine neue Klasse die gegebenen Bausteine für ihre Implementierung benutzt. Die Komposition im engeren Sinne geschieht dann auf der Objektebene. Zwei Fragen stehen dabei im Vordergrund: Wie werden die Objekte untereinander verbunden? Wie kommunizieren die Objekte miteinander? Die verwendeten Mechanismen sind recht unterschiedlich, wie die folgenden Beispiele zeigen:

- Die wohl einfachste Variante ist es, die beteiligten Objekte in Programmvariablen zu halten und darüber eine Programmschicht zu legen, die die Kommunikation mit und zwischen diesen Objekten über Methodenaufrufe abwickelt; d.h. die Bausteine werden durch beliebigen, von vornherein nicht vorstrukturierten Programmtext verbunden.
- Einige Bausteine sind darauf vorbereitet, über bestimmte Protokolle oder Softwarekanäle miteinander zu kommunizieren. Ein Beispiel dazu werden wir im Rahmen der verteilten Programmierung mit den Client-Server-Bausteinen in Kap. 7 behandeln.
- Es gibt Bausteine, die von vornherein das Zusammenstecken mit anderen Bausteinen unterstützen. Die Kommunikation zwischen den Bausteinen bleibt dabei verborgen. Dies gilt beispielsweise für einige der Ströme, die wir in diesem Kapitel kennen lernen werden.
- Wirken mehrere Bausteine zusammen, geschieht die Kommunikation zwischen ihnen oft über spezielle Kommunikationsmechanismen. Beispielsweise basiert die Kommunikation in vielen Programngerüsten (z.B. zur Konstruktion graphischer Bedienoberflächen; vgl. Kap. 5) auf

einer sogenannten Ereignissteuerung, deren Realisierung vom Gerüst selbst unterstützt wird.

Diese unterschiedlichen Kompositionstechniken zeigen zum einen die Flexibilität, die Programmiersprachen fürs Realisieren von Bausteinen bieten; zum anderen erschweren sie aber auch die gemeinsame Nutzung von Bausteinen unterschiedlicher Arten.

**Beschreibung von Bausteinen.** Für das erfolgreiche Arbeiten mit wiederverwendbaren Bausteinen spielen zwei weitere Aspekte eine wichtige Rolle: Wie sind Bausteine beschrieben? Wie kann man Bausteine finden, die bei der Lösung einer softwaretechnischen Aufgabe hilfreich sein könnten? Auf beide Fragen gibt es erste, aber sicherlich keine allgemein befriedigenden Antworten. Wir werden hier kurz auf die erste Frage eingehen.

Relativ gut untersucht sind Beschreibungstechniken für unabhängige Bausteine. Unabhängige Bausteine hängen weder in Verhalten noch im Zustand von anderen Softwareteilen ab und gestatten üblicherweise nur eine Benutzung über Methoden. Dementsprechend lassen sie sich als *abstrakte Datentypen* beschreiben. Unter einem abstrakten Datentyp versteht man im Allgemeinen eine Menge von Elementen (bzw. Objekten) zusammen mit den Operationen, die für die Elemente der Menge charakteristisch sind. Die Elemente und die Operationen bilden dabei eine Einheit, deren Verhalten unabhängig von dem Verhalten anderer Typen und unabhängig von der Implementierung spezifiziert werden kann.

Das grundlegende Prinzip, Daten und Operationen zusammenzufassen, stand Pate bei der Entwicklung des Klassenkonzepts in der objektorientierten Programmierung. Dementsprechend sind Klassen gut geeignet, um abstrakte Datentypen zu implementieren. Sie stellen einerseits dem Benutzer eine klar definierte Schnittstelle zur Verfügung und erlauben es andererseits, Implementierungsaspekte zu verbergen. Eine unabhängige Klasse kann man dadurch beschreiben, dass man angibt, welche Zustände ihre Objekte annehmen können, welche Ergebnisse die Methoden der Klasse liefern und wie sie den Zustand ändern. Solche Beschreibungen bestehen üblicherweise aus einer natürlich-sprachlichen Erläuterung des Zwecks der Klasse und ihrer möglichen Zustände, den Methodensignaturen sowie einer informellen Beschreibung der Wirkungsweise der Methoden. Mittlerweile stehen aber auch recht ausgereifte formale Spezifikationstechniken zur Verfügung.

Schwieriger ist es, Bausteine zu beschreiben, die nur im Zusammenwirken mit anderen Bausteinen eine Aufgabe erfüllen können und deren Methoden nicht nur auf den Baustein selbst wirken, sondern auch den Zustand anderer Bausteine verändern. Eine Methode für die Beschreibung eng kooperierender Bausteine werden wir in Kap. 5 behandeln.

### 4.1.2 Überblick über die Java-Bibliothek

Bausteine für programmierungsnah, häufig vorkommende Standardaufgaben werden üblicherweise als fester Bestandteil der Programmierumgebung einer Sprache in Form standardisierter Bibliotheken angeboten. Dies gilt insbesondere für objektorientierte Sprachen, bei denen die Bibliotheken durch die Erweiterbarkeit der Bausteine mittels Vererbung noch an Bedeutung gewonnen haben. Bei vielen modernen objektorientierten Sprachen ist es sogar erklärtes Entwurfsziel, die Sprache – also die Anzahl der Sprachkonstrukte – möglichst klein zu halten und alle mit Mitteln der Sprache formulierbaren Bestandteile in die Standardbibliothek zu verlagern. Dadurch ergibt sich ein enges Zusammenwirken zwischen der Programmiersprache und der Standardbibliothek. Bei Smalltalk geht es so weit, dass selbst der Typ Boolean als eine Bibliotheksklasse realisiert wird. In Java werden wir die Beziehung zwischen Sprache und Standardbibliothek am Beispiel der Fehlerbehandlung im nächsten Abschnitt kennen lernen.

Um eine exemplarische Vorstellung von Standardbibliotheken zu vermitteln, stellen wir im Folgenden einen Ausschnitt der Java-Bibliothek kurz vor. Abbildung 4.1 zeigt einen Teil der Pakete. Dabei entspricht jeder Baumknoten mit Ausnahme des Wurzelknotens einem Paket. Der Paketname wird wie in Kap. 2 beschrieben gebildet. Als Überblick und Suchhilfe erläutern wir zu

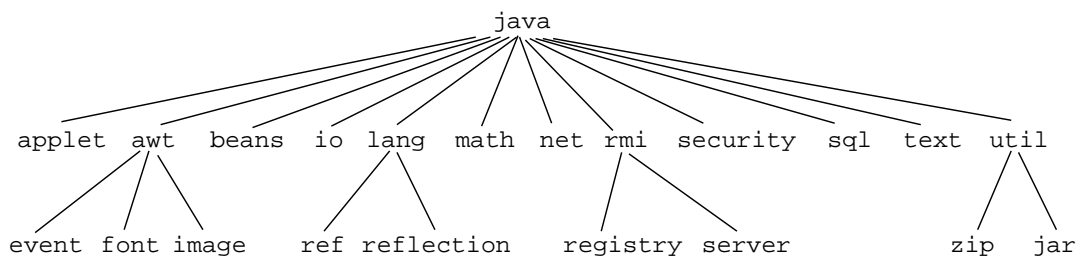


Abbildung 4.1: Ausschnitt aus der Java-Bibliothek

jedem der angegebenen Pakete kurz den Anwendungsbereich der darin enthaltenen Klassen:

- `java.applet` enthält Klassen, um Java-Programme in WWW-Seiten zu integrieren. Diese *Applets* genannten Programme werden beim Laden der WWW-Seite im Browser gestartet und können mit dem WWW-Server kommunizieren, von dem sie geladen wurden. Applets erlauben es, WWW-Seiten mit umfangreicher Funktionalität auszustatten.
- `java.awt` enthält Klassen zur Realisierung graphischer Bedienoberflächen (siehe Kap. 5). Die zugehörigen Klassen zur Ereignissteuerung sind im Paket `java.awt.event` zusammengefasst. Darüberhinaus gibt es eine Reihe von untergeordneten Paketen für spezielle Aspek-

*Applet*



te; beispielsweise enthält `java.awt.images` Klassen zur Darstellung von Bildern.

- `java.beans` fasst die Klassen zusammen, mit denen die Realisierung sogenannter Java-Beans unterstützt wird. Java-Beans sind Softwarekomponenten, die man in speziellen Werkzeugen, sogenannten Builder Tools, graphisch manipulieren und zusammensetzen kann (vgl. Kap. 8).
- `java.io` enthält Klassen für die Ein- und Ausgabe (siehe Abschn. 4.3).
- `java.lang` ist das Standardpaket. Es enthält u.a. die Klassen `Object`, `String` und `StringBuffer`, die Wrapper-Klassen für alle Basisdatentypen, Schnittstellen zum Betriebs- und Laufzeitsystem sowie die vordefinierten Klassen für die Fehlerbehandlung (siehe Abschn. 4.2) und für die parallele Programmierung (siehe Kap. 6).
- `java.math` enthält u.a. Klassen zur Programmierung mit ganzen und Dezimalzahlen beliebiger Genauigkeit.
- `java.net` stellt Klassen zur Verfügung, mit deren Hilfe man über sogenannte Socket-Verbindungen innerhalb von Netzwerken kommunizieren kann (siehe Kap. 7).
- `java.rmi` stellt Klassen zur Verfügung, um über sogenannten entfernten Methodenaufruf (engl. `remote method invocation`) innerhalb von Netzwerken zu kommunizieren (siehe Kap. 7). Die Klassen für die Verwaltung von Namen sind im Paket `java.rmi.registry` zusammengefasst; `java.rmi.server` enthält Klassen, die beim entfernten Methodenaufruf auf der Serverseite benötigt werden.
- `java.security` bietet Klassen an, mit denen man recht detailliert regeln kann, was der aktuelle Ausführungsstrang eines Programmes darf und welche Aktionen ihm verboten sind. Darüber hinaus enthält das Paket Klassen zur Verschlüsselung und zum Signieren von Objekten.
- `java.sql` realisiert die Programmierschnittstelle, mit der Datenbanken von Java-Programmen aus über SQL-Anweisungen angesprochen werden können.
- `java.text` enthält Klassen für eine einheitliche Behandlung von Text, Datumsangaben, Zahlen und Nachrichten. Ziel dabei ist es, diese Angaben unabhängig von den Formaten zu machen, die in den unterschiedlichen natürlichen Sprachen der Welt verwendet werden. Die Anpassung an die jeweils gültige Landessprache soll dann auf diesem einheitlichen Standard aufsetzen.
- `java.util` realisiert viele allgemein nützliche Klassen, insbesondere:

- Wichtige Datentypen: Listen, Stapel, Bitfelder, Hashtabellen, Dictionaries und vieles andere mehr.
- Klassen zur Behandlung von Zeit- und Kalenderdaten.

Der Überblick ist in zweifacher Hinsicht unvollständig. Zum einen wurden in der obigen Pakethierarchie einige untergeordnete Pakete weggelassen. Zum anderen gibt es zwei weitere Paketverzeichnisse, die standardmäßig zur Verfügung stehen: Das Verzeichnis `javax` enthält Erweiterungen der Standardbibliothek (das `x` steht für Extension), insbesondere die beiden Pakete `javax.swing` und `javax.accessibility`. Diese und ihre untergeordneten Pakete stellen eine effizientere, flexiblere und umfangreichere Bibliothek zur Realisierung graphischer Bedienoberflächen bereit. Die im Verzeichnis `org.omg` enthaltenen Pakete bieten eine Programmierschnittstelle zu CORBA-Anwendungen (CORBA ist eine standardisierte Architektur zur Realisierung verteilter, heterogener Anwendungen; vgl. Abschn. 7.1.2).

## 4.2 Ausnahmebehandlung mit Bausteinen

Ausnahmebehandlung hat nur indirekt etwas mit bausteinorientierter Programmierung zu tun. Wir betrachten sie hier aus drei Gründen.

1. Die Klassen, mit deren Objekten in Java die Ausnahmebehandlung gesteuert wird, bieten wohl das einfachste Beispiel für eine erweiterbare Bausteinhierarchie.
2. Sie zeigen das enge Zusammenspiel zwischen Sprache und Bibliothek.
3. Im Übrigen benötigen wir eine genauere Kenntnis der Ausnahmebehandlung im Rest des Kurses.

Dieser Abschnitt behandelt zunächst die Bibliotheksbausteine, mit denen Java die Ausnahmebehandlung unterstützt, und fasst dann das Zusammenwirken von Sprachkonstrukten und Bibliotheksbausteinen zur Ausnahmebehandlung zusammen.

### 4.2.1 Eine Hierarchie von einfachen Bausteinen

Immer wenn die Ausführung eines Ausdrucks oder eine Anweisung in Java abrupt terminiert (vgl. Abschn. 1.3.1.3), wird ein Ausnahmeobjekt erzeugt, das die Ausnahme klassifiziert und die nachfolgende Ausnahmebehandlung steuert. In Java werden verschiedene Arten von Ausnahmen unterschieden:

1. Ausnahmesituationen, auf die der Benutzer im Rahmen des Programms nur wenig Einfluss nehmen kann. Dazu gehört insbesondere das Erschöpfen der zugewiesenen Maschinenressourcen (kein Speicher mehr verfügbar, Kellerüberlauf).

2. Ausnahmen, die durch defensives Programmieren vermeidbar sind; beispielsweise der Versuch, die null-Referenz zu dereferenzieren, der Zugriff mit einem unzulässigen Index auf ein Feld oder eine Zeichenreihe oder der Aufruf einer Methode mit falschen Parametern.
3. Ausnahmen, die im Allgemeinen nicht vermeidbar sind, aber vom Programm behandelt werden können bzw. sollten. Dazu gehören Zugriffe auf programm-externe Ressourcen, z.B. auf das Dateisystem oder das Netzwerk (eine Datei kann von einem anderen Benutzer gelöscht oder blockiert worden sein, so dass auf sie nicht mehr zugegriffen werden kann; eine Netzwerkverbindung kann gestört sein). In diese Kategorie fallen auch vom Programmierer eingeführte Ausnahmen, die für seine spezifische Anwendung benötigt werden. Dazu gehören z.B. die Überprüfung von Eingabefeldern auf zulässige Eingabedaten etc.

Die Grundbausteine zur Beschreibung von Ausnahmen sind in einer Klassenhierarchie im Paket `java.lang` enthalten. Sie spiegeln die obige Einteilung der Ausnahmearten wider. Abb. 4.2 stellt die wesentlichen Ausnahmeklassen dar. Alle Klassen zur Beschreibung von Ausnahmeobjekten erben von der Klasse `Throwable`. Jedes `Throwable`-Objekt enthält eine Feh-

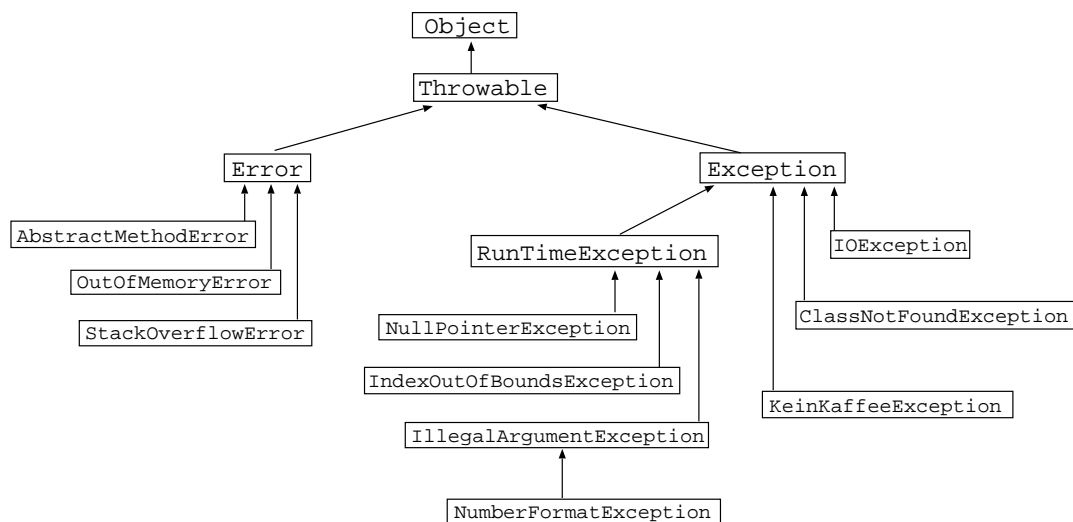


Abbildung 4.2: Ausnahmhierarchie

meldung, die mit Hilfe der Methode `getMessage` ausgelesen werden kann und bietet Methoden an, um den Inhalt des Laufzeitkellers zu drucken (`printStackTrace`). Ausnahmen der ersten Art werden in Java *Fehler* genannt; ihre Objekte sind von der Klasse `Error` abgeleitet. Ausnahmen der zweiten Art, also solche, die bei typischen Programmierfehlern auftreten, sind als Unterklassen von `RuntimeException` realisiert. Bei allen anderen

Ausnahmen sieht Java vor, dass sie direkt oder indirekt<sup>2</sup> von `Exception` abgeleitet werden. Insbesondere kann jeder Programmierer selbst Ausnahmeklassen als Subklassen von `Exception` deklarieren. In Abb. 4.2 haben wir dies durch die Klasse `KeinKaffeeException` angedeutet, die etwa von einer Kaffeemaschinensteuerung ausgelöst werden könnte, wenn das vorhandene Kaffeepulver für den anstehenden Aufbrühvorgang nicht mehr ausreicht. Deren Deklaration könnte beispielsweise wie folgt aussehen:

```
public class KeinKaffeeException extends Exception {
    private float restMenge;
    KeinKaffeeException( float kaffeeMenge ) {
        restMenge = kaffeeMenge;
    }
    public float getRestMenge() { return restMenge; }
}
```

Ausnahmeobjekte dieser Klasse merken sich auch die vorhandene Restmenge an Kaffeepulver. An der Programmstelle, an der eine Ausnahme vom Typ `KeinKaffeeException` behandelt wird, lässt sich diese Menge mittels der Methode `getRestMenge` auslesen und dementsprechend z.B. entscheiden, ob man Kaffeepulver nachfüllen soll oder lieber weniger Kaffee zubereitet. Auf diese Weise können Ausnahmeobjekte Informationen von der Stelle, an der die Ausnahme ausgelöst wurde, zu der Stelle transportieren, an der sie behandelt werden kann.

Die meisten Ausnahmeklassen definieren weitgehend unabhängige Bausteine mit einer sehr einfachen Funktionalität. Diese Klassen als Bausteine zu bezeichnen, wirkt schon beinahe ein wenig überzogen. Dies hängt wohl auch damit zusammen, dass man Ausnahmeobjekte nicht zu Objektgeflechten zusammenbaut, sondern sie meistens nur als Informationsobjekte durchreicht. Andererseits stellen sie Programmteile mit einem hohen Wiederverwendungsgrad dar und bieten eine flexible Technik, um die Vielfalt möglicher Ausnahmesituationen<sup>3</sup> zu strukturieren, beherrschbar zu machen und bzgl. neuer Situationen spezialisieren zu können.

## 4.2.2 Zusammenspiel von Sprache und Bibliothek

Ein typisches Problem der Ausnahmebehandlung in größeren Programmsystemen besteht darin, dass die Ausnahmen oft nicht direkt an der Programmstelle behandelt werden können, an der sie auftreten. Betrachten wir beispielsweise ein in Schichten aufgebautes Programmsystem, bei dem jede Schicht nur die Methoden der direkt darunterliegenden Schicht benutzen darf. Tritt in der untersten Schicht eine Ausnahme auf, die nur in der obersten

<sup>2</sup>Nicht via `RuntimeException`.

<sup>3</sup>Die Java-Bibliothek enthält allein mehr als hundert `Exception`-Klassen.

Schicht behandelt werden kann – beispielsweise, weil der Anwender des Programms gefragt werden muss, – muss die Information über die Ausnahme in geeigneter Weise von den Zwischenschichten weitergereicht werden. Dies geschieht mit Hilfe der Ausnahmeobjekte und bestimmten Sprachkonstrukten. Wie deren Zusammenspiel in Java aussieht, ist Gegenstand dieses Abschnitts.

Ähnlich wie in vergleichbaren Sprachen (z.B. C++) lässt sich in Java die Funktionsweise der Fehlerbehandlung in drei Aspekte aufteilen:

1. Auslösen von Ausnahmen;
2. Abfangen und Behandeln von Ausnahmen;
3. Weiterreichen von Ausnahmen über Methodengrenzen hinweg.

Aus Programmiersicht steht hinter dem ersten Aspekt die Frage: Wie kann es in meinem Programm zu einer Ausnahmesituation, also einer abrupten Terminierung eines Ausdrucks oder einer Anweisung kommen? Sinnvollerweise unterscheidet man drei Fälle:

- (a) Der Aufruf einer Methode oder eines Konstruktors terminiert abrupt.
- (b) Die Auswertung eines elementaren Ausdrucks terminiert abrupt (z.B. Division durch null, Dereferenzierung der null-Referenz, unzulässige Typkonvertierung).
- (c) Mittels der throw-Anweisung wird explizit eine Ausnahme ausgelöst.

Den Fall (c) haben wir nochmals anhand des kleinen Programmfragments von Abb. 4.3 veranschaulicht, in dem die Methode zum Filter-Füllen die KeinKaffee-Ausnahme auslöst, wenn der Kaffeespeicher weniger als die benötigte Menge an Kaffeepulver enthält.

```
public class KaffeeMaschine {
    private KaffeeSpeicher speicher;
    ...

    void fuellenFilter( float benoetigteMenge )
                        throws KeinKaffeeException {
        float restMenge;

        restMenge = speicher.messenFuellung();
        if( restMenge < benoetigteMenge )
            throw new KeinKaffeeException( restMenge );
        ...
    }
}
```

Abbildung 4.3: Erzeugen der KeinKaffee-Ausnahme

In Abschn. 1.3.1.3, S. 41, wurde beschrieben, dass der zur throw-Anweisung gehörende Ausdruck immer zu einem Ausnahmeobjekt ausgewertet

werden muss. Dies können wir nun dahingehend präzisieren, dass der Typ des Ausdrucks immer ein Subtyp von `Throwable` sein muss. Damit haben wir ein erstes Beispiel für das Zusammenspiel von Sprache und Standardbibliothek: Die angegebene Kontextbedingung der Sprache ist mittels eines Typs der Bibliothek formuliert.

Für das Abfangen und Behandeln von Ausnahmen gibt es in Java die `try`-Anweisung (vgl. Abschn. 1.3.1.3). Tritt im `try`-Block eine Ausnahme vom Typ  $E$  auf, wird die erste `catch`-Klausel ausgeführt, deren zugehöriger Ausnahmetyp ein Supertyp von  $E$  ist. Hier wird also der Typ der Ausnahme zur Fallunterscheidung bei der Behandlung verwendet. Die vorkommenden Ausnahmetypen müssen Subtypen von `Throwable` sein.

Beim Weiterreichen von Ausnahmen sind im Wesentlichen zwei Fälle zu unterscheiden:

1. Das abrupt terminierende Programmstück (Ausdruck oder Anweisung) liegt in einer `try`-Anweisung. Dann wird entweder die erzeugte Ausnahme von der `try`-Anweisung abgefangen oder die `try`-Anweisung terminiert selbst abrupt (und es stellt sich die Frage, ob die `try`-Anweisung Teil einer sie umfassenden `try`-Anweisung ist).
2. Das Programmstück ist in keiner `try`-Anweisung enthalten. Dann terminiert der Aufruf der umfassenden Methode  $m$  abrupt. D.h. die Ausführung von  $m$  terminiert und kehrt zur Aufrufstelle von  $m$  zurück, wobei das Ausnahmeobjekt  $A$  quasi als Ergebnis an den Aufrufer übertragen wird. Mit  $A$  als Ausnahmeobjekt terminiert dann auch der Ausdruck abrupt, von dem  $m$  aufgerufen wurde.

Genauso wie man den Typ des regulären Ergebnisses einer Methode in der Methodensignatur deklarieren muss, fordert Java die Deklaration der Ausnahmen, die eine Methode möglicherweise auslöst (vgl. Abschn. 2.1.2, S. 79). Dies geschieht in der `throws`-Deklaration, die aus dem Schlüsselwort `throws` und einer Liste von Typnamen besteht. Alle aufgelisteten Typen müssen Subtypen von `Throwable` sein. Präziser gesagt, wird Folgendes verlangt:

Ist  $T$  der Typ einer Ausnahme, die im Methodenrumpf möglicherweise ausgelöst und nicht behandelt wird, dann muss  $T$  oder ein Supertyp von  $T$  in der `throws`-Deklaration erscheinen.

Von der Deklarationspflicht ausgenommen sind alle Ausnahmetypen, die von `Error` oder `RuntimeException` abgeleitet sind.

Überschreibt eine Methode  $m$  aus  $S$  eine Methode  $m$  aus  $T$ , muss es zu jedem Ausnahmetyp der `throws`-Deklaration von  $m$  aus  $S$  einen Supertyp<sup>4</sup> in der `throws`-Deklaration von  $m$  aus  $T$  geben; d.h. die `throws`-Deklaration von  $m$  aus  $T$  muss mehr Ausnahmen zulassen. Die `throws`-Deklarationen verhalten sich also kovariant (vgl. Unterabschn. 3.2.2.2, S. 174).

<sup>4</sup>Zur Erinnerung: Die Sub- und Supertyprelation sind reflexiv, d.h. jeder Typ ist Subtyp und Supertyp von sich selbst.

## 4.3 Stromklassen: Bausteine zur Ein- und Ausgabe

Ein Stromobjekt ermöglicht es, sukzessive eine Folge von Daten zu lesen bzw. zu schreiben. Dafür stellt es Methoden zum Lesen bzw. Schreiben zur Verfügung. In vielen Betriebssystemen werden Ströme für die Behandlung von Ein- und Ausgaben eingesetzt (man denke beispielsweise an *standard in* und *standard out* in Unix). Aus Sicht der Bausteindiskussion bieten objektorientierte Ströme zwei sehr interessante Aspekte. Zum einen sind sie Beispiele für eigenständige Bausteine: in ihrer Schnittstelle verwenden sie nur ihre Supertypen, selbstdeklarierte oder vordefinierte Typen. Zum anderen können Ströme bausteinartig zusammengesetzt werden (ähnlich dem Pipe-Mechanismus in Unix): Beispielsweise kann ein Bytestrom aus einer Datei lesen und die gelesenen Bytes an einen Strom weitergeben, der die Bytes zu Zahlenfolgen zusammenfasst.

Dieser Abschnitt bietet im ersten Teil eine kurze Einführung in das Stromkonzept. Der zweite Teil gibt dann eine Übersicht über die Stromklassen der Java-Bibliothek und geht insbesondere auf die Serialisierung von Objektgeflechten ein.

### 4.3.1 Ströme: Eine Einführung

Anhand mehrerer einfacher Stromklassen wird im Folgenden in das Stromkonzept und das bausteinartige Zusammensetzen von Strömen eingeführt. Dazu konzentrieren wir uns auf Eingabeströme, also Ströme zum Lesen von Eingaben. Für Ausgabeströme gilt Entsprechendes. In unseren Beispielen betrachten wir Ströme, aus denen man nacheinander einzelne Zeichen auslesen kann:

```
interface CharEingabeStrom {
    int read() throws IOException;
}
```

Bei der Signatur der Methode `read` halten wir uns an die Konventionen von Java: Im Normalfall gibt die `read`-Methode das nächste Zeichen des Eingabestroms zurück. Wenn der Eingabestrom keine weiteren Zeichen enthält, liefert sie das Ergebnis `-1`. Wenn auf den Eingabestrom nicht zugegriffen werden kann (beispielsweise weil er seine Daten aus einer Netzverbindung zu lesen versucht, die unterbrochen ist), terminiert die `read`-Methode abrupt mit einer `IOException`. Um Überschneidungen beim Ergebnis zwischen dem Normalfall und dem Erreichen des Stromendes zu vermeiden, hat `read` den Ergebnistyp `int` (die Werte vom Typ `char` entsprechen den Zahlen von 0 bis 65535 im Typ `int`; vgl. Abb. 1.6, S. 27). Das gelesene Zeichen erhält man durch eine Typkonvertierung nach `char`.

Eingabeströme beziehen die bereitgestellte Eingabe aus unterschiedlichen *Quellen* (entsprechend für Ausgabeströme): Sie lesen beispielsweise aus Dateien, aus Datenstrukturen (z.B. Feldern) oder aus einem Netzwerk; sie bekommen die Eingabe interaktiv von Benutzern oder anderen Programmen; sie beziehen die Eingabe von einem oder mehreren anderen Eingabeströmen. Wie wir sehen werden, ermöglicht es die letztgenannte Variante, Ströme bausteinartig hintereinander zu stecken.

Das Grundkonzept von Strömen betrachten wir anhand der folgenden drei Eingabeströme:

- einen String-Leser, der sukzessive die Zeichen aus einem String-Objekt liest, das ihm bei der Erzeugung übergeben wurde;
- einen Filter, der alle Zeichen eines Stromes in Großbuchstaben umwandelt;
- einen Filter, der die Umlaute und das „ß“ in einem übergebenen Zeichenstrom mittels zweier Buchstaben transliteriert.

Der Konstruktor des String-Lesers speichert die Zeichen des Strings in einem Feld, aus dem sie von der `read`-Methode sukzessive ausgelesen werden:

```
public class StringLeser implements CharEingabeStrom {
    private char[] dieZeichen;
    private int    index = 0;

    public StringLeser( String s ) {
        dieZeichen = s.toCharArray();
    }

    public int read() {
        if( index == dieZeichen.length )
            return -1;
        else return dieZeichen[index++];
    }
}
```

Der Großbuchstaben-Filter liest die Zeichen aus einem Eingabestrom und konvertiert alle Kleinbuchstaben in Großbuchstaben; dafür benutzt er die Methode `toUpperCase` aus der Bibliotheksklasse `java.lang.Character`:



```

public class GrossBuchstabenFilter
    implements CharEingabeStrom {
    private CharEingabeStrom eingabeStrom;

    public GrossBuchstabenFilter( CharEingabeStrom cs ) {
        eingabeStrom = cs;
    }
    public int read() throws IOException {
        int z = eingabeStrom.read();
        if( z == -1 )
            return -1;
        else return Character.toUpperCase( (char)z );
    }
}

```

Ähnlich arbeitet der Umlaut-Filter (vgl. Abb. 4.4). Da die Umlaute bzw. das „ß“ aber jeweils durch zwei Zeichen ersetzt werden, muss er das zweite Zeichen bis zum nächsten read-Aufruf im Attribut `naechstesZ` zwischenspeichern. Abbildung 4.4 zeigt auch die Anwendung der Stromklassen – vor allem, wie sie bausteinartig aneinander gehängt werden können: Auf den String-Leser wird der Umlaut-Filter gesteckt, der dann wiederum dem Großbuchstaben-Filter als Eingabe dient. (Was passiert, wenn man die beiden Filter vertauscht?)

Zwar dient das Programmbeispiel im Wesentlichen dazu, das Konzept von Strömen zu demonstrieren; man sollte aber bereits im Ansatz die Kombinationsmöglichkeiten von Strömen erkennen.

```

public class UmlautSzFilter implements CharEingabeStrom {
    private CharEingabeStrom eingabeStrom;
    private int naechstesZ = -1;

    public UmlautSzFilter( CharEingabeStrom cs ) {
        eingabeStrom = cs;
    }

    public int read() throws IOException {
        if( naechstesZ != -1 ) {
            int z = naechstesZ;
            naechstesZ = -1;
            return z;
        } else {
            int z = eingabeStrom.read();
            if( z == -1 ) return -1;
            switch( (char)z ) {
                case '\u00C4': naechstesZ = 'e'; return 'A';
                case '\u00D6': naechstesZ = 'e'; return 'O';
                case '\u00DC': naechstesZ = 'e'; return 'U';
                case '\u00E4': naechstesZ = 'e'; return 'a';
                case '\u00F6': naechstesZ = 'e'; return 'o';
                case '\u00FC': naechstesZ = 'e'; return 'u';
                case '\u00DF': naechstesZ = 's'; return 's';
                default:      return z;
            }
        }
    }
}

public class Main {
    public static void main(String[] args) throws IOException {
        String s = new String(
            "\u00C4neas opfert den G\u00F6ttern edle \u00D6le,\nauf "
            +"da\u00DF \u00FCberall das \u00DCbel sich \u00E4ndert.");

        CharEingabeStrom cs;
        cs = new StringLeser( s );
        cs = new UmlautSzFilter( cs );
        cs = new GrossBuchstabenFilter( cs );

        int z = cs.read();
        while( z != -1 ) {
            System.out.print( (char)z );
            z = cs.read();
        }
        System.out.println("");
    }
}

```

Abbildung 4.4: Umlaut-Filter und Testrahmen für die Stromklassen

**Adaption von Bausteinen.** Wie das Beispiel zeigt, sorgen die Konstrukto-  
ren für das Verbinden der Ströme. Dies funktioniert allerdings nur, wenn die  
Ströme den geeigneten Typ haben. Besitzt man eine Stromklasse, die im We-  
sentlichen das Gewünschte leistet, aber nicht den passenden Typ hat, muss  
man eine *Adapterklasse* schreiben.

*Adapterklasse*

Wir demonstrieren dieses Vorgehen an einem sehr einfachen Beispiel. Angenommen, wir wollen in der main-Methode von Abb. 4.4 die Zeichen nicht aus einem String, sondern aus einer Datei lesen. Dann bräuchten wir einen Datei-Leser, der Subtyp von `CharEingabeStrom` ist. Das Bibliothekspaket `java.io` von Java stellt zwar einen Datei-Leser unter dem Namen `FileReader` mit einer geeigneten `read`-Methode zur Verfügung; aber `FileReader` ist kein Subtyp von `CharEingabeStrom`. In diesem Fall ist die Adapterklasse, die wir `DateiLeser` nennen wollen, sehr einfach. Mit der von `FileReader` geerbten Implementierung implementiert sie die Schnittstelle `CharEingabeStrom`:

```
public class DateiLeser extends FileReader
    implements CharEingabeStrom {
    public DateiLeser( String s ) throws IOException {
        super(s);
    }
}
```

Der `String`-Parameter des Konstruktors bezeichnet dabei den Namen der Datei, aus der gelesen werden soll. In der Praxis reicht es meistens nicht aus, nur den Typen der Klasse zu adaptieren; vielmehr sind häufig auch Methoden anzupassen oder neue hinzuzufügen. Dies ändert aber nichts am Prinzip: Adapterklassen passen die Schnittstelle existierender Bausteine an, damit diese von Klienten benutzt werden können, die eine andere Schnittstelle erwarten.

### 4.3.2 Ein Baukasten mit Stromklassen

Java bietet in seiner reichhaltigen Bibliothek von Stromklassen für die verschiedenen Ein- und Ausgabeformate sowie -schnittstellen eine jeweils angepasste Funktionalität. Dieser Abschnitt bietet eine Übersicht über die Stromklassen, demonstriert ihre Anwendung und behandelt etwas genauer die Serialisierung von Objekten. Ziel des Abschnitts ist es zum einen, die Vererbungs- und Typbeziehungen zwischen eigenständigen Bausteinen an einem realistischen Beispiel zu studieren, und zum anderen, wichtige programmier-technische Kenntnisse zu vermitteln.

#### 4.3.2.1 Javas Stromklassen: Eine Übersicht

Java bietet in seiner Bibliothek eine Vielzahl von Stromklassen an. Die meisten dieser Klassen befinden sich im Paket `java.io`. Im Wesentlichen sind diese Klassen in vier Hierarchien organisiert:

1. Reader-Klassen sind Eingabeströme auf Basis des Typs `char`.
2. Writer-Klassen sind Ausgabeströme auf Basis des Typs `char`.
3. `InputStream`-Klassen sind Eingabeströme auf Basis des Typs `byte`.
4. `OutputStream`-Klassen sind Ausgabeströme auf Basis des Typs `byte`.

Alle von Reader abgeleiteten Klassen unterstützen:

- das Lesen eines einzelnen Zeichens: `int read()`,
- das Lesen mehrerer Zeichen in ein `char`-Feld: `int read(char[])` bzw. `int read(char[],int,int)`,
- das Überspringen einer Anzahl von Zeichen: `long skip(long)`,
- eine Abfrage, ob der Strom zum Lesen bereit ist: `boolean ready()`,
- das Schließen des Eingabestroms: `void close()`,
- Methoden zum Markieren und Zurücksetzen des Stroms.

Alle von Writer abgeleiteten Klassen unterstützen:

- das Schreiben eines einzelnen Zeichens: `void write(int)`,
- das Schreiben von Zeichen eines `char`-Feldes: `void write(char[])` bzw. `void write(char[],int,int)`,
- das Schreiben mehrerer Zeichen eines Strings: `void write(String)` bzw. `void write(String,int,int)`,
- die Ausgabe ggf. im Strom gepufferter Zeichen: `void flush()`,
- das Schließen des Ausgabestroms: `void close()`.

Bei der Angabe der obigen Signaturen haben wir der Einfachheit halber den Ausnahmetyp `IOException` für geworfene Ausnahmen nicht mit aufgeführt.

Die von `InputStream` bzw. `OutputStream` abgeleiteten Klassen leisten Entsprechendes für den Basisdatentyp `byte` (demgemäß entfallen die Methoden zur Ausgabe von Strings). Die Abbildungen 4.5, 4.6 und 4.7 geben eine Übersicht über die vier Stromklassenhierarchien. Schnittstellen besitzen

dabei einen gestrichelten Rahmen, abstrakte Klassen einen durchbrochenen und Klassen einen durchgezogenen (vgl. auch die Erläuterungen zu Abb. 3.5). Aus Darstellungsgründen wurden die Klassennamen abgekürzt: Das Kürzel „R“ am Ende eines Klassennamens steht für Reader, „W“ für Writer, „IS“ für InputStream, „OS“ für OutputStream.

Wir betrachten zunächst die Reader-Hierarchie und erläutern ausgewählte Klassen. Der InputStreamReader liest Bytes aus einem InputStream und

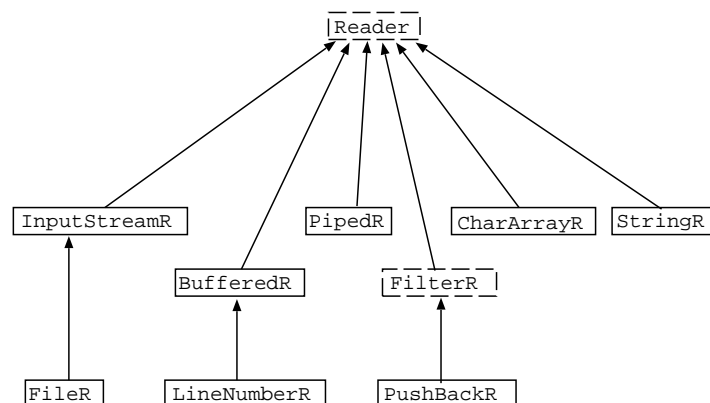


Abbildung 4.5: Die Readerklassen realisieren char-Eingabeströme

konvertiert diese nach `char`, also in Unicode-Zeichen. Die verwendete Konvertierungsfunktion kann beim Konstruktoraufbau eingestellt werden. Ein File-Reader ist ein InputStream-Reader, der die Eingabebytes aus einer Datei liest. File-Reader leisten das Gleiche wie InputStream-Reader, die einen `FileInputStream` als Quelle haben.

Ein Buffered-Reader hat einen anderen Reader als Quelle (insofern verhält er sich wie ein Filter; bei den Stream-Klassen sind die gepufferten Ströme auch von der entsprechenden Filterklasse abgeleitet; vgl. Abb. 4.7). Er liest immer mehrere Zeichen auf einmal und speichert diese zwischen. Dadurch wird es vermieden, für jedes einzelne Zeichen immer neu auf eine Quelle zuzugreifen. Durch das Zwischenschalten eines Buffered-Readers kann beispielsweise die Effizienz beim Zugriff auf Dateien oder Netzwerke erheblich gesteigert werden. Die Klasse `BufferedReader` stellt zusätzlich zu den Methoden der Klasse `Reader` eine Methode zum Lesen einer ganzen Zeile zur Verfügung: `String readLine()`. Der `LineNumberReader` zählt automatisch die aktuelle Zeilennummer mit und bietet Methoden, um sie abzufragen und zu setzen.

Ein `PipedReader` liest aus einem `PipedWriter` und ermöglicht damit das Aneinanderhängen von Aus- und Eingabeströmen. Die abstrakte Klasse `FilterReader` unterstützt die Implementierung von Eingabefiltern, d.h. von Eingabeströmen, die einen anderen Strom als Quelle haben (vgl. die Beispiele in Abschn. 4.3.1). Ein Beispiel für einen Filter ist der `PushBackReader`,

der es mittels der Methode `unread` gestattet, gelesene Zeichen in den Strom zurückzuschieben. Die Klassen `CharArrayReader` und `StringReader` benutzen `char`-Felder bzw. Strings als Quelle.

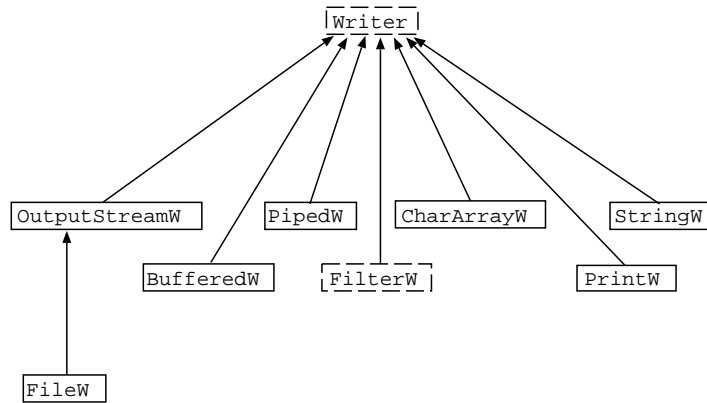


Abbildung 4.6: Die Writerklassen realisieren char-Ausgabeströme

Alles für die Reader Gesagte gilt entsprechend für die Writer-Klassen. Die Klasse `PrintWriter` stellt darüber hinaus Methoden zum Ausdrucken der Basisdatentypen in lesbarer Form zur Verfügung. Das folgende Beispiel dient zur Demonstration wichtiger Reader- und Writer-Klassen. Die Methode `schreiben` ermöglicht es, eine Zeichenreihe in eine Datei zu schreiben; die Methode `lesen` liest den Inhalt einer Datei und gibt ihn in Form eines Strings zurück.

```

import java.io.*;

public class DateiZugriff {
    public static void schreiben( String dateiname, String s )
        throws IOException {
        PrintWriter out;
        out = new PrintWriter( new FileWriter( dateiname ) );
        out.print( s );
        out.close();
    }

    public static String lesen(String dateiname)
        throws FileNotFoundException, IOException {
        BufferedReader in =
            new BufferedReader(new FileReader(dateiname));
        StringBuffer inputstr = new StringBuffer("");
        String line;
        line = in.readLine();
        while (line != null) {
            inputstr = inputstr.append(line);
            inputstr = inputstr.append("\n");
            line = in.readLine();
        }
    }
}
  
```

```

    }
    in.close();
    return inputstr.toString();
}
}

public class DateiZugriffTest {
    public static void main( String[] argf ){
        String s;
        try {
            s = DateiZugriff.lesen( argf[0] );
        } catch( FileNotFoundException e ){
            System.out.println("Can't open "+ argf[0] );
            return;
        } catch( IOException e ){
            System.out.println("IOException reading "+ argf[0] );
            return;
        }
        try {
            DateiZugriff.schreiben("ausgabeDatei",s);
        } catch( IOException e ){
            System.out.println("Can't open "+ argf[0] );
        }
    }
}

```

Mit den von `InputStream` und `OutputStream` abgeleiteten Klassen bietet Java eine ganze Familie von Klassen zur Realisierung von byte-Strömen (vgl. Abb. 4.7). Da die beiden byte-Strom-Hierarchien in allen wesentlichen Aspekten der Reader- bzw. Writer-Hierarchie entsprechen, begnügen wir uns hier mit einer kurzen Erörterung der zusätzlichen Typen. Die Schnittstellen `DataInput` und `DataOutput` umfassen Methoden zum Lesen bzw. Schreiben aller Basisdatentypen. Die Schnittstellen `ObjectInput` und `ObjectOutput` enthalten darüber hinaus u.a. Methoden zum Lesen bzw. Schreiben von Objekten (vgl. Unterabschn. 4.3.2.2). Für jede der Schnittstellen gibt es eine Klasse, die die entsprechende Funktionalität bereitstellt. Außerdem implementiert die Klasse `RandomAccessFile` die Schnittstellen `DataInput` und `DataOutput`. Sie ermöglicht es – wie bei gängigen Dateisystemen üblich – lesend und schreibend auf eine Datei zuzugreifen.

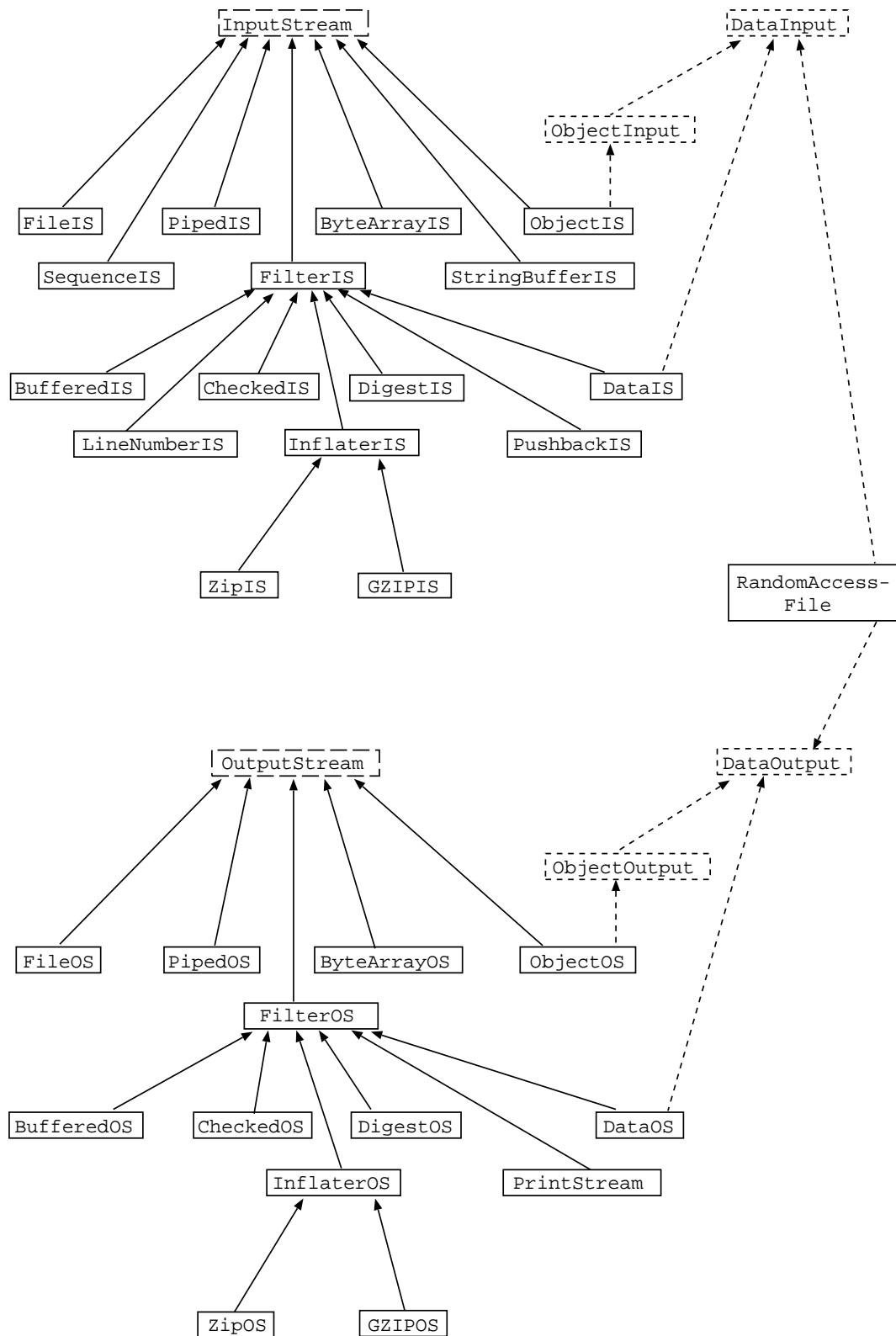


Abbildung 4.7: Javas bytebasierte Stromklassen



**Nachbetrachtung.** Die Stromklassen von Java sind grundsätzlich<sup>5</sup> eigenständige Bausteine. Durch Anwendung von Abstraktion konnten für viele Stromklassen gemeinsame Supertypen definiert werden. Diese Supertypen sind die Voraussetzung dafür, verschiedenste Stromklassen miteinander kombinieren zu können. Die dadurch entstehende Abhängigkeit einer Stromklasse zum Supertyp ist also gewünscht und notwendig. Sie führt auch nicht dazu, dass die Beschreibung einer Stromklasse  $S$  erschwert wird: Um  $S$  zu beschreiben, muss man zwar auch alle seine Parametertypen beschreiben; außer Standardtypen können dies aber nur Supertypen von  $S$  sein. Da die Supertypen von  $S$  weniger Funktionalität besitzen als  $S$  selbst, ist die Beschreibung der Supertypen quasi in der Beschreibung von  $S$  enthalten.

Die Kombinierbarkeit der Stromklassen ist im Wesentlichen durch die Konstruktoren festgelegt. Die Verbindung zwischen den Strömen wird dabei gekapselt. Trotz der Kapselung lassen sich mittels Vererbung relativ leicht weitere Stromklassen realisieren, die zusätzliche Kombinationsmöglichkeiten bieten und die Funktionalität erweitern oder auf spezielle Erfordernisse zuschneiden.

Die Stromklassen demonstrieren aber auch Unzulänglichkeiten von Java. Die Existenz ähnlicher Hierarchien für char- und byte-Leser bzw. -Schreiber führt zu einem erhöhten Lernaufwand und erschwert damit die Benutzung.

#### 4.3.2.2 Ströme von Objekten

Das Lesen und Schreiben von Basisdaten ist einfach. Ihre Werte besitzen eine eindeutige Darstellung, die von anderen Werten unabhängig ist. Dies gilt nicht für Objekte und Objektreferenzen. Objektreferenzen sind zwar eindeutig innerhalb eines laufenden Prozesses, sie besitzen aber keine explizite Darstellung und ihre implizite Darstellung als Speicheradresse verliert außerhalb des Prozesses ihre Gültigkeit. Objekte lassen sich auch nicht ausschließlich durch ihren Zustand repräsentieren. Denn erstens muss man Objekte unterscheiden können, die den gleichen Zustand haben, und zweitens bleibt das Problem der Darstellung von Objektreferenzen bestehen, da der Zustand häufig Referenzen auf andere Objekte umfasst.

Dieser Abschnitt erläutert, wie Java die Ein- und Ausgabe von Objekten bzw. Objektgeflechten mittels Strömen unterstützt. Die Ein- und Ausgabe von Objekten ist wichtig, um Objekte zwischen mehreren Prozessen austauschen zu können (beispielsweise über eine Netzverbindung) und um Objekte, die in einem Programmlauf erzeugt wurden, für spätere Programmläufe verfügbar zu machen. Im zweiten Fall spricht man auch davon, Objekte *persistent* zu machen, d.h. ihre Lebensdauer über die Programmlaufzeit hinaus zu verlängern.

Persistenz

<sup>5</sup>Einige Stromklassen benutzen Typen, die nicht in `java.lang` enthalten sind, aber als Standardtypen betrachtet werden können.

**Ausgabe von Objektgeflechten.** Um genauer zu verstehen, was Aus- und Eingabe von Objekten bedeutet, betrachten wir folgende doppelt verkettete Liste `ll` von Zeichenreihen<sup>6</sup>:

```
LinkedList ll = new LinkedList();
StringBuffer s = new StringBuffer("Sand");
ll.add("Sich "); ll.add("mit "); ll.add(s); ll.add("alen ");
ll.add("im "); ll.add(s); ll.add(" aalen");
```

Was könnte es heißen, `ll` in eine Datei ausgeben zu wollen? Soll nur das `LinkedList`-Objekt ausgegeben werden oder auch alle referenzierten Entry-Objekte, die implizit von der `add`-Methode erzeugt werden (vgl. Abb. 2.8, S. 100, zur Realisierung doppelt verketteter Listen)? Oder möchte man auch alle referenzierten `String`-Objekte mit ausgeben? Sinnvollerweise wird man `ll` und alle über Referenzen von `ll` aus erreichbaren Objekte ausgeben; denn nur so ist es möglich, die Liste `ll` wieder vollständig zu restaurieren. Bei einer derartigen Ausgabe bekommt jedes referenzierte Objekt eine Nummer. Objektreferenzen werden dann mittels der Objektnummer kodiert.

Da Objektgeflechte Zirkularitäten enthalten können (wie im Beispiel zwischen den Entry-Objekten) und da Programmierer häufig die Struktur der Objekte benutzter Klassen nicht kennen und auf deren Attribute ggf. nicht zugreifen dürfen, ist die Realisierung der Ausgabe von Objektgeflechten eine nicht-triviale Aufgabe. Angenehmerweise gibt es in Java die Objektströme, die diese Aufgabe für den Benutzer lösen. Beispielsweise lassen sich `ll` und alle referenzierten Objekte mit folgenden Anweisungen in eine Datei ausgeben:

```
OutputStream os = new FileOutputStream("speicherndeDatei");
ObjectOutputStream oos = new ObjectOutputStream( os );
oos.writeObject( ll );
```

Man beachte, dass dabei mehrfach referenzierte Objekte, beispielsweise das `StringBuffer`-Objekt der Variablen `s`, nur einmal ausgegeben werden. (Dies gilt für alle Objekte mit der Ausnahme von `String`-Objekten. Deshalb haben wir im Beispiel statt `String` die Klasse `StringBuffer` benutzt, die veränderliche Zeichenreihen implementiert.)

**Eingabe von Objektgeflechten.** Das Einlesen und Restaurieren von Objektgeflechten geht programmtechnisch ganz entsprechend wie das Ausgeben. Ein kleiner Unterschied ergibt sich daraus, dass aus den eingelesenen Daten neue Objekte erzeugt werden müssen. Dazu benutzt der Eingabestrom die Methode `forname` der Klasse `java.lang.Class`. Findet er die benötigte Klasse nicht, wird eine `ClassNotFoundException` ausgelöst:

<sup>6</sup>Würde man nach Ausführung aller `add`-Anweisungen sämtliche Elemente der Liste hintereinander auf der Systemausgabe ausgeben, ergäbe sich folgender Text: Sich mit Sandalen im Sand aalen.

```

LinkedList inll;
InputStream is = new FileInputStream("speicherndeDatei");
ObjectInputStream ois = new ObjectInputStream( is );
try {
    inll = (LinkedList)ois.readObject();
} catch( ClassNotFoundException exc ) {
    System.out.println("Klasse zu Eingabeobjekt fehlt");
}

```

Obige Anweisungen lesen ein LinkedList-Objekt und alle von ihm direkt und indirekt referenzierten Objekte aus der Datei `speicherndeDatei` und bauen das entsprechende Objektgeflecht im einlesenden Prozess auf. Man beachte, dass das resultierende Objektgeflecht zwar die gleiche Geflechtstruktur und die gleichen Objektzustände wie das ausgegebene Objektgeflecht aufweist, dass es aber nur eine Kopie des ausgegebenen Objektgeflechts ist. Genau genommen werden also nicht Objekte ausgegeben und eingelesen, sondern nur ihre Geflecht- und Zustandsinformation. Anhand des Beispiels in Abb. 4.8 lässt sich das gut studieren. Die linke Abbildungshälfte zeigt ein Objektgeflecht bestehend aus den Objekten *o1* bis *o5*<sup>7</sup>, das von den Variablen *a*,

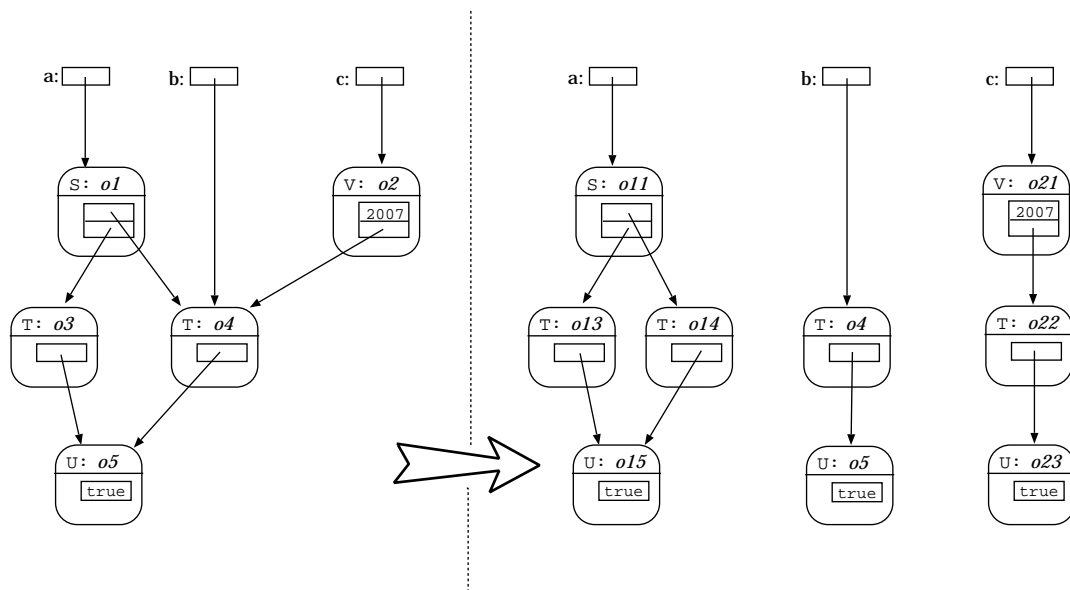


Abbildung 4.8: Vor- und nach Ausgabe und Einlesen von *a* und *c*

*b* und *c* in angegebener Weise referenziert wird. Die rechte Abbildungshälfte zeigt den Objektspeicher, nachdem zunächst das von *a*, dann das von *c* referenzierte Objekt ausgegeben und im Anschluss wieder eingelesen wurde. Sowohl *a* als auch *c* referenzieren danach neu erzeugte Geflechte (Objekte

<sup>7</sup>Die Schreibweise „X: *o*“ in der Kopfzeile der dargestellten Objekte bedeutet, dass das Objekt *o* vom Typ X ist.

*o11* bis *o15* bzw. *o21* bis *o23*), die nach dem Einlesen aber keine gemeinsamen Objekte mehr enthalten. Die Variable *b* verweist nach wie vor auf das Objekt *o4* und indirekt auf *o5*.

**Zur Funktionsweise von Objektströmen.** Objektströme sind ein schönes Beispiel für Bausteine mit einfacher Schnittstelle und leistungsfähiger Funktionalität. Objektströme müssen bei der Ausgabe mehrfach referenzierte Objekte erkennen und Zirkularitäten auflösen können. Dazu vergeben sie für jedes Objekt eine Nummer und verwalten die Zuordnung der ausgegebenen Objekte zu deren Nummern in einer Tabelle. Steht beim Durchlauf durch das Objektgeflecht ein weiteres Objekt *obj* zur Ausgabe an, wird zunächst anhand der Tabelle geprüft, ob *obj* bereits ausgegeben wurde. Ist das der Fall, wird nur die zugehörige Objektnummer herausgeschrieben. Andernfalls erhält *obj* eine Nummer, wird in die Tabelle eingetragen und dann mit seinen Attributwerten in den Ausgabestrom geschrieben. Auf diese Weise werden alle im Geflecht erreichbaren Objekte behandelt und *der Reihe nach* ausgegeben; man spricht deshalb auch vom *Serialisieren* von Objektgeflechten. Für das Einlesen gilt Entsprechendes.

Damit ein Objektstrom das von einem Objekt erreichbare Geflecht vollständig ausgeben kann, muss er auch Zugriff auf die privaten Attribute des Geflechts haben. Würde man Objektströmen diesen Zugriff immer automatisch gewähren, könnten Objektströme missbraucht werden, um geschützte Implementierungsteile auszuforschen. Deshalb sieht Java einen Mechanismus vor, mit dem deklariert werden muss, ob eine Klasse serialisierbar sein soll. Eine Klasse ist genau dann serialisierbar, wenn sie ein Subtyp des Schnittstellentyps `java.io.Serializable` ist. D.h. serialisierbare Klassen müssen entweder `Serializable` „implementieren“ oder einen Supertyp besitzen, der bereits Subtyp von `Serializable` ist; die Eigenschaft, serialisierbar zu sein, wird also vom Supertyp an den Subtyp weitergegeben. Da der Schnittstellentyp `Serializable` weder Attribute noch Methoden besitzt, braucht eine Klasse, die `Serializable` implementieren soll, diesen Typ nur zusätzlich in die `implements`-Liste aufzunehmen. Bei dem Versuch, ein Objekt einer nicht serialisierbaren Klasse auszugeben, wird eine `NotSerializableException` ausgelöst.



## Selbsttestaufgaben

### Aufgabe 1: Sichtbarkeit

Notieren Sie in dem folgenden Programmcode für alle durch eine vorangestellte Nummer gekennzeichneten Anweisungen, ob der Zugriff auf das verwendete Attribut bzw. die verwendete Methode bzw. die verwendete Klasse erlaubt ist oder nicht. Begründen Sie jeweils Ihre Antwort.

```
package A;

public class A1 {
    private int i = 0;
    protected int j = 1;

    private void am1() {
1)    System.out.println("Dies ist eine private Methode" + i);
    }
    protected void am2() {
2)    System.out.println(
        "Dies ist eine Methode mit Modifikator 'protected'" + j);
    }

    boolean isEqualTo(A1 a) {
3)    if ( i == a.i ) return true;
        return false;
    }
    public boolean compareA1toB1 (B1 b) {
4)    if ( i == b.i ) return true;
        return false;
    }
}

class B1 {
    public int i = 2;
    int j = 3;
    void bm() {
        A1 a = new A1();
        A1 a2 = new A1();
5)    a.j = 10;
6)    a.isEqualTo(a2);
7)    a.am1();
    }
}
```

---

```

package B;
import A.*;

class D1 {
    private int i = 9;

    class D2 {
        private int i = 0;
    }

    D2 createD2Element () {
        return new D2();
    }
}

public class C1 extends A1 {
    void cm( A1 a, A1 a2, C1 c) {
8)    B1 b = new B1();
9)    D1 d = new D1();
10)   D1.D2 d2 = d.createD2Element();
11)   a.i = 10;
12)   c.i = 15;
13)   c.j = 20;
14)   a.isEqualTo(a2);
15)   c.am2();
    }
}

```

### Aufgabe 2: Werktag als Subklasse der Klasse Wochentag

Leiten Sie von der Klasse `Wochentag` aus der Selbsttestaufgabe 1 der Kurs-einheit 2 eine Klasse `Werktag` mit den unten angegebenen Eigenschaften ab, deren Objekte nur die Tage von Montag bis Samstag aufzählen! (D.h. nach einem Samstag folgt ein Montag und vor einem Montag liegt ein Samstag.)

- Die Klasse `Werktag` soll keine weiteren Attribute vereinbaren.
- Sorgen Sie dafür, dass die Methoden `naechsterTag()`, `vorhergehenderTag()` und `setTag(int i)` nur Werkstage berücksichtigen!

### Aufgabe 3: Von Mehrfachvererbung zu Einfachvererbung

Es gibt oftmals Situationen beim Entwurf von Klassenhierarchien, in denen man Mehrfachvererbung einsetzen möchte. Diese Entwürfe können dann nicht direkt in Java umgesetzt werden. In dieser Aufgabe sollen Sie ein Muster kennen lernen, mit dem man Klassenhierarchien, die Mehrfachvererbung benutzen, in Klassenhierarchien umwandelt, die nur Einfachvererbung benutzen.

Im Folgenden wird beschrieben, wie die Umwandlung vorzunehmen ist. Seien *S1*, *S2* und *C* Klassen und es gelte *C extends S1, S2* (was in Java natürlich nicht erlaubt ist):

- Erstellen Sie ein Interface *S2\_Interface*, das die Methodenschnittstelle von *S2* enthält!
- *C* erhält ein Attribut *s2* vom Typ *S2*.
- *C* erbt von *S1* und implementiert *S2\_Interface*.
- Die Methodenimplementierungen des Interface *S2\_Interface* in *C* rufen die entsprechenden Methoden des *s2*-Attributes von *C* auf.
- Passen Sie den Konstruktor von *C* so an, dass das notwendige *S2*-Objekt des *s2*-Attributes zu einem geeigneten Zeitpunkt erzeugt wird!

#### Aufgaben:

- a) Wandeln Sie das unten angegebene Programm gemäß dem Muster um! Dabei gehören die Klassen *Item*, *Process*, *Ball* zur Implementierung eines Computerspiels, bei dem sich Bälle in einem Spielfeld bewegen und dabei an Wände stoßen, an denen sie zurückgeworfen werden. Bälle und Wände sind Instanzen von Klassen, die aus einer gemeinsamen Klasse *Item* abgeleitet wurden. Bälle sind außerdem aktive Objekte (Prozesse). Alle Klassen aktiver Objekte sind aus einer gemeinsamen Klasse *Process* abgeleitet.

```
class Item {
    private int pos_x;
    private int pos_y;

    public void draw() { /*...*/; }
    public void move(int dx, int dy) {
        pos_x += dx;
        pos_y += dy;
        //....
    }
}
```



```

class Process {
    private int state = 0; // 0 = passiv, 1 = active

    public void activate() { state = 1; }
    public void passivate() { state = 0; }
    public int getState() { return state; }
}

class Ball extends Item, Process {
    private int radius;

    public void setRadius(int r) { radius = r; }
    public int getRadius() { return radius; }
}

```

- b) Zeigen Sie an einem Beispiel, dass es ungünstig ist, die Methoden der Klasse S2 direkt in der Klasse C zu implementieren anstatt sie an ein internes Objekt vom Typ S2 zu delegieren. Begründen Sie Ihre Antwort.

#### Aufgabe 4: Überschreiben und Überladen

Welche Ausgaben liefert das im Folgenden angegebene Programm? Erläutern Sie, warum es zu diesen Ausgaben kommt. Erläutern Sie dazu insbesondere, wie die aktuell auszuführende Methode ausgewählt wird.

```

class A { }

class B extends A {
    void m(A a) {
        System.out.println("m in Klasse B mit Argument vom Typ A");
    }
    void m(B b) {
        System.out.println("m in Klasse B mit Argument vom Typ B");
    }
    void m(C c) {
        System.out.println("m in Klasse B mit Argument vom Typ C");
    }
}

class C extends B {
    void m(A a) {
        System.out.println("m in Klasse C mit Argument vom Typ A");
    }
    void m(B b) {
        System.out.println("m in Klasse C mit Argument vom Typ B");
    }
    void m(C c) {
        System.out.println("m in Klasse C mit Argument vom Typ C");
    }
}

```

```
void m() {
    System.out.println("m in Klasse C ohne Parameter");
}

}

public class Test {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        C c = new C();

        b.m(a);
        b.m(b);
        b.m(c);

        c.m(a);
        c.m(b);
        c.m(c);

        B db = new B();
        C cc = new C();
        db = cc;

        db.m(a);
        db.m(b);
        db.m(c);

        db.m(db);
        db.m( (C) db);

    }
}
```

### Aufgabe 5: Statisches und dynamisches Binden

Gegeben sei das folgende Java-Programm:

```
class A {
    int i = 1;
    int m() { return i * 100; }
    int n() { return i + m(); }
    void k() { System.out.println("" + this); } /* 6 */
}

class B extends A {
    int i = 10;
    int m() { return i * 1000; }
    int test2() { return n(); }
}
```

```

class Test {
    int test1() {
        A[] ar = { new A(), new B() };
        return ar[0].i + ar[0].m() + ar[1].i + ar[1].m();
    }

    public static void main(String[] argv) {
        System.out.println(new A().m());           /* 1 */

        A a = new B();
        System.out.println(a.m());                 /* 2 */

        System.out.println(a.i);                   /* 3 */

        System.out.println(new B().test2());       /* 4 */

        System.out.println(new Test().test1());    /* 5 */
    }
}

```

Bitte beantworten Sie die folgenden Fragen und begründen Sie Ihre Antworten! Alle Fragen beziehen sich auf das gegebene Programm.

#### Aufgaben:

- a) Wie viele und welche Attribute hat ein Objekt vom Typ B?
- b) In objektorientierten Sprachen muss man zwischen dem statischen und dynamischen Typ eines Ausdrucks unterscheiden. Welchen statischen Typ hat der Ausdruck `ar[1]` in der zweiten Zeile des Methodenrumpfes von `test1` und welchen dynamischen Typ hat er bei Programmausführung?
- c) Welchen Wert liefert der Ausdruck `ar[1].i` in der zweiten Zeile des Methodenrumpfes von `test1` bei Programmausführung?
- d) Welcher Wert wird an der mit „1“ gekennzeichneten Stelle ausgegeben?
- e) Welcher Wert wird an der mit „2“ gekennzeichneten Stelle ausgegeben?
- f) Welcher Wert wird an der mit „3“ gekennzeichneten Stelle ausgegeben?
- g) Welcher Wert wird an der mit „4“ gekennzeichneten Stelle ausgegeben?
- h) Welcher Wert wird an der mit „5“ gekennzeichneten Stelle ausgegeben?
- i) Stellen Sie sich vor, das gegebene Programm könnte um beliebige Klassen erweitert werden. Welchen statischen Typ hat und welchen dynamischen Typ kann die Variable `this` an der mit „6“ gekennzeichneten Stelle haben?

## Musterlösungen zu den Selbsttestaufgaben

### Aufgabe 1: Sichtbarkeit

Im folgenden Programmcode ist hinter jeder markierten Anweisung angegeben, ob der Zugriff auf das verwendete Attribut bzw. die verwendete Methode oder Klasse erlaubt ist. Dahinter wird jeweils eine Begründung angegeben.

```
package A;

public class A1 {
    private int i = 0;
    protected int j = 1;

    private void am1() {
1)      System.out.println("Dies ist eine private Methode" + i);
                                   // OK: Auf private Elemente, die in der eigenen
    }                                   // Klasse deklariert sind, darf zugegriffen werden.

    protected void am2() {
2)      System.out.println(
            "Dies ist eine Methode mit Modifikator 'protected'" + j);
                                   // OK: Auf protected Elemente, die in der eigenen
    }                                   // Klasse deklariert sind, darf zugegriffen werden.

    boolean isEqualTo(A1 a) {
3)      if ( i == a.i ) return true;
                                   // OK: Der Zugriff auf a.i ist erlaubt, da aus einer
    return false;                 // Klasse auf alle Attribute und Methoden von
    }                             // Objekten (auch privaten) zugegriffen werden kann,
                                   // die denselben Typ besitzen wie die betrachtete
                                   // Klasse.

    public boolean compareA1toB1 (B1 b) {
4)      if ( i == b.i ) return true;
                                   // OK: Der Zugriff auf b.i ist erlaubt, da i mit
    return false;                 // dem Modifikator public versehen wurde.
    }
}

class B1 {
    public int i = 2;
    int j = 3;
    void bm() {
        A1 a = new A1();
        A1 a2 = new A1();

5)      a.j = 10;                 // OK: Der Zugriff auf a.j ist erlaubt, da j mit dem
                                   // Modifikator protected versehen wurde, der den
                                   // default access innerhalb von Paketen einschliesst.

6)      a.isEqualTo(a2);         // OK: isEqualTo besitzt default access und B1
    }
```

```

// gehoert zum selben Paket wie A1.

7)    a.am1();           // FALSCH: am1 ist als private deklariert. Daher kann
                        // von ausserhalb von A1 nicht auf am1 zugegriffen
                        // werden.
    }
}

-----

package B;
import A.*;

class D1 {
    private int i = 9;

    class D2 {
        private int i = 0;
    }

    D2 createD2Element () {
        return new D2();
    }
}

public class C1 extends A1 {
    void cm( A1 a, A1 a2, C1 c)  {
8)    B1 b = new B1();         // FALSCH: Die Deklaration einer Variablen vom Typ B1
                        // ist nicht moeglich, da B1 nur im Paket A sichtbar
                        // ist (default access).

9)    D1 d = new D1();         // OK: Die Deklaration einer Variablen vom Typ D1
                        // ist moeglich, da D1 im Paket B mit default access
                        // deklariert ist.

10)   D1.D2 d2 = d.createD2Element();
                        // OK: Die innere Klasse D2 von D1 ist aufgrund
                        // des default access im Paket B ueber die
                        // Punktnotation sichtbar. Ebenso hat createD2Element
                        // default access und ist somit in C1 zugreifbar.

11)   a.i = 10;               // FALSCH: i ist in A1 als private deklariert und
                        // daher von C1 aus nicht zugreifbar.

12)   c.i = 15;               // FALSCH: i ist in A1 als private deklariert und
                        // daher selbst von der abgeleiteten Klasse C1 aus
                        // nicht zugreifbar.

13)   c.j = 20;               // OK: j ist in A1 als protected deklariert und daher
                        // in allen abgeleiteten Klasse, also auch C1,
                        // zugreifbar.

```

```
14)    a.isEqualTo(a2);    // FALSCH: isEqualTo besitzt nur default access und
                        // ist deshalb von ausserhalb des Pakets A nicht
                        // zugreifbar.

15)    c.am2();            // OK: am2 ist in A1 als protected deklariert und
                        // daher in allen abgeleiteten Klasse, also auch C1,
                        // zugreifbar.

    }
}
```

### Aufgabe 2: Werktag als Subklasse der Klasse Wochentag

Die folgende Implementierung der Klasse `Werktag` erfüllt die geforderten Eigenschaften. Bei der Implementierung der Klasse `Werktag` ist darauf zu achten, dass der Zugriff auf das private Attribut `tag` der Klasse `Wochentag` nicht möglich ist. Die Implementierung muss daher die Methoden der Klasse `Wochentag` benutzen, um die überschreibenden Methoden geeignet zu implementieren. Wird dazu `setTag` benutzt, so muss eine entsprechende Ausnahmebehandlung implementiert werden.

```
class Werktag extends Wochentag {

    public void setTag(int i) throws KeinTagException {
        if (i < 0 || i > 5) throw new KeinTagException();
        else super.setTag(i);
    }

    public void naechsterTag() {
        super.naechsterTag();
        if(getTag() == 6) super.naechsterTag();
    }

    public void vorhergehenderTag() {
        super.vorhergehenderTag();
        if(getTag() == 6) super.vorhergehenderTag();
    }
}
```

**Aufgabe 3: Von Mehrfachvererbung zu Einfachvererbung**

- a) Wenn man die Transformation nach dem angegebenen Muster durchführt, erhält man das folgende Programm:

```

class Item {
    private int pos_x;
    private int pos_y;

    public void draw() { /*...*/; }
    public void move(int dx, int dy) {
        pos_x += dx;
        pos_y += dy;
        //....
    }
}

interface Process_Interface {
    public void activate();
    public void passivate();
    public int getState();
}

class Process implements Process_Interface {
    private int state = 0; // 0 = passiv, 1 = active

    public void activate() { state = 1; }
    public void passivate() { state = 0; }
    public int getState() { return state; }
}

public class Ball extends Item implements Process_Interface {
    private Process process;
    private int radius;

    public Ball () { process = new Process(); }
    public void activate() { process.activate(); }
    public void passivate() { process.passivate(); }
    public int getState() { return process.getState(); }

    public void setRadius(int r) { radius = r; }
    public int getRadius() { return radius; }
}

```

- b) Im folgenden Beispiel gibt es 2 Arten aktiver Objekte, Bälle und Kugeln. Die zugehörigen Klassen `Ball` und `Bowl` implementieren beide das Interface `Process_Interface`, nehmen aber inverse Zustände für „aktiv“ und „passiv“ an. Daher ist die frühere Implementierung über Mehrfachvererbung nicht richtig umgesetzt worden. Dort wäre `Bowl` genauso wie auch `Ball` von `Process` abgeleitet worden und die Zustände

„aktiv“ und „passiv“ wären durch denselben Wert von `state` repräsentiert worden (auslesbar durch `getState`).

```
class Item {
    private int pos_x;
    private int pos_y;

    public void draw() { /*...*/; }
    public void move(int dx, int dy) {
        pos_x += dx;
        pos_y += dy;
        //....
    }
}

interface Process_Interface {
    public void activate();
    public void passivate();
    public int getState();
}

class Ball extends Item implements Process_Interface {
    private int state = 0; // 0 = passiv, 1 = active
    private int radius;

    public void activate() { state = 1; }
    public void passivate() { state = 0; }
    public int getState() { return state; }

    public void setRadius(int r) { radius = r; }
    public int getRadius() { return radius; }
}

class Bowl extends Item implements Process_Interface {
    private int state = 1; // 1 = passiv, 0 = active
    private int radius;
    private boolean flexible_material = true;

    public void activate() { state = 0; }
    public void passivate() { state = 1; }
    public int getState() { return state; }

    public void setRadius(int r) { radius = r; }
    public int getRadius() { return radius; }
    //....
}
```

Dieser Fehler könnte nicht vorkommen, wenn alle ehemals von `Process` abgeleiteten Klassen ein Attribut vom Typ `Process` besäßen wie in a) demonstriert und die Methodenaufrufe für die Methoden



aus `Process.Interface` an dieses Attribut delegieren würden. Dann wäre auch in jedem Fall eine identische Implementierung gewährleistet.

#### Aufgabe 4: Überschreiben und Überladen

Das Programm führt zu unten stehender Ausgabe. Dabei sind die Leerzeilen in der Musterlösung nur der Übersicht halber eingefügt worden. Sie werden nicht vom Programm erzeugt.

```

m in Klasse B mit Argument vom Typ A      // ausgegeben durch b.m(a)
m in Klasse B mit Argument vom Typ B      // ausgegeben durch b.m(b)
m in Klasse B mit Argument vom Typ C      // ausgegeben durch b.m(c)

m in Klasse C mit Argument vom Typ A      // ausgegeben durch c.m(a)
m in Klasse C mit Argument vom Typ B      // ausgegeben durch c.m(b)
m in Klasse C mit Argument vom Typ C      // ausgegeben durch c.m(c)

m in Klasse C mit Argument vom Typ A      // ausgegeben durch db.m(a)
m in Klasse C mit Argument vom Typ B      // ausgegeben durch db.m(b)
m in Klasse C mit Argument vom Typ C      // ausgegeben durch db.m(c)

m in Klasse C mit Argument vom Typ B      // ausgegeben durch db.m(db)
m in Klasse C mit Argument vom Typ C      // ausgegeben durch db.m((C)db)

```

Im Folgenden betrachten wir das Programm und seine Besonderheiten etwas genauer.

1. Zu den Aufrufen `b.m(a)` ; `b.m(b)` ; und `b.m(c)` ;

Da `b` vom statischen und dynamischen Typ `B` ist, werden die `m`-Methoden aus `B` aufgerufen. Die jeweils passende Methode `m` wird durch Auflösung der Überladung zur Übersetzungszeit anhand der Typen der aktuellen und formalen Parameter ausgewählt.

2. Zu den Aufrufen `c.m(a)` ; `c.m(b)` ; und `c.m(c)` ;

Hier gilt dasselbe wie in 1. Da `c` vom statischen und dynamischen Typ `C` ist, werden die `m`-Methoden aus `C` aufgerufen. Die jeweils passende Methode `m` wird durch Auflösung der Überladung zur Übersetzungszeit anhand der Typen der aktuellen und formalen Parameter ausgewählt.

3. Zu den Aufrufen `db.m(a)` ; `db.m(b)` ; und `db.m(c)` ;

Da die Überladung zur Übersetzungszeit aufgelöst wird, wird diesen drei Aufrufen jeweils die Methode mit dem genau passenden Parametertyp der Klasse `B` zugeordnet. Da `db` aber zur Laufzeit ein Objekt vom Typ `C` enthält (dynamischer Typ `C`), werden diese Methoden zur Laufzeit durch die entsprechenden Methoden der Klasse `C` überschrieben. Es werden also die drei `m`-Methoden der Klasse `C` aufgerufen.

4. Zu den Aufrufen `db.m(db);` und `db.m((C) db);`

Der erste Aufruf zeigt sehr schön, dass Überladung zur Übersetzungszeit statisch aufgelöst wird, wohingegen Überschreibung anhand des dynamischen Typs erst zur Laufzeit aufgelöst wird. Anhand des deklarierten Typs `B` von `db` wird die Methode mit der Signatur `void m(B b)` der Klasse `B` als Methode mit der spezifischsten Methodensignatur ermittelt (Auflösen der Überladung). Zur Laufzeit hat `db` aber den dynamischen Typ `C`. Deshalb wird zur Laufzeit die Methode mit der Signatur `void m(B b)` der Klasse `C` ausgeführt, die die Methode mit dieser Signatur der Klasse `B` überschreibt. Deshalb kommt es zur Ausgabe von „m in Klasse C mit Argument vom Typ B“.

Der zweite Aufruf zeigt den Effekt von expliziten Typecasts. Diesmal wird bei der statischen Auflösung der Überladung wegen des Typecasts `(C) db` für `db` der Typ `C` zugrundegelegt. Deshalb kommt es jetzt zur Ausgabe „m in Klasse C mit Argument vom Typ C“.

Im Detail ist der Prozess zur statischen Auflösung von Überladung und zur dynamischen Auflösung von Überschreibung im Buch

- Gosling/Joy/Steele  
The Java Language Specification  
Kapitel 15.11, S.323 ff

beschrieben. Etwas knapper auch in

- Arnold/Gosling  
The Java Programming Language Specification, 3rd Edition  
Kapitel 15.12.2.5, S.447 ff

### Aufgabe 5: Statisches und dynamisches Binden

- a) Ein Objekt vom Typ `B` besitzt zwei Attribute. Das `int`-Attribut `i` aus der Klasse `B` und das `int`-Attribut `i` aus der Klasse `A`. Weitere Erläuterung: Sei `x` eine Referenz auf ein `B`-Objekt vom statischen Typ `B`. Da Attribute statisch gebunden werden, greift man mit `x.i` auf die Instanz des Attributes `i` der Klasse `B` zu. Will man auf die Instanz des Attributes `i` aus der Klasse `A` zugreifen, so muss `x` auf den Typ `A` gecastet werden: `((A) x).i`
- b) Die lokale Variable `ar` speichert eine Referenz auf ein Array vom statischen Typ `A[]`, d.h. das Array speichert Referenzen auf Objekte vom statischen Typ `A`. Somit hat der Ausdruck `ar[1]` den statischen Typ `A`. Bei der Initialisierung des Arrays wird an der Stelle mit dem Index 1 ein Objekt vom Typ `B` gespeichert. Dies ist möglich, da `B` Subtyp von `A` ist. Der dynamische Typ des Ausdrucks `ar[1]` ist also `B`.

- c) Der Ausdruck `ar[1].i` liefert den Wert 1, da der statische Typ von `ar[1]` `A` ist und Attributzugriffe statisch gebunden werden.
- d) An der mit `/* 1 */` gekennzeichneten Stelle wird 100 ausgegeben. Die Implementierung der Methode `m` aus der Klasse `A` wird ausgeführt. Diese Methode multipliziert den Wert des Attributs `i` der aktuellen `A`-Instanz mit 100, liefert also  $1 * 100 = 100$  zurück.
- e) An der mit `/* 2 */` gekennzeichneten Stelle wird 10000 ausgegeben. Die Variable ist vom statischen Typ `A`, referenziert aber ein `B`-Objekt. Der Methodenaufruf wird also dynamisch an das `m` aus `B` gebunden. `i` in `m` wird statisch an das `i` aus `B` gebunden, dessen Wert für jede `B`-Instanz 10 ist. Die Methode `m` aus `B` gibt also den Wert  $10 * 1000$  zurück.
- f) An der mit `/* 3 */` gekennzeichneten Stelle wird 1 ausgegeben, da `a` vom statischen Typ `A` ist. Also wird das `i` aus `A` ausgegeben.
- g) An der mit `/* 4 */` gekennzeichneten Stelle wird 10001 ausgegeben.

**Ausführung der `test2`-Methode:**

Die `test2`-Methode aus `B` wird ausgeführt. Im Rumpf der `test2`-Methode wird die Methode `n` aus `A` ausgeführt, da `B` `n` von `A` erbt. Der implizite Parameter bei der Ausführung von `n` ist jedoch das `B`-Objekt. (#)

`test2` liefert den von `n` erhaltenen Rückgabewert selbst unverändert zurück. Dieser wird dann auf der Systemausgabe ausgegeben.

**Ausführung von `n` innerhalb der `test2`-Methode:**

Da `n` in `A` deklariert ist, ist der statische Typ des impliziten Parameters von `n` `A`, d.h. das `i` im Rumpf der Methode `n` wird statisch an das `i` aus `A` gebunden, sodass `i` den Wert 1 besitzt. `n` gibt daher den um 1 erhöhten Rückgabewert des Aufrufs von `m` in `n` zurück. Der Methodenaufruf von `m` in `n` wird dynamisch an das `m` aus `B` gebunden, da der implizite Parameter von `n` dynamisch vom Typ `B` ist (s.o., (#)).

**Ausführung von `m` innerhalb der Methode `n`:**

Das Attribut `i` in `m` aus `B` wird statisch an das `i` aus `B` gebunden, das für jede `B`-Instanz den Wert 10 besitzt. Dieser Aufruf von `m` innerhalb der Methode `n` liefert also  $10 * 1000 = 10000$  zurück (siehe auch e)).

**Endergebnis:**

Damit liefert der Aufruf von `n` den Wert  $1 + 10000 = 10001$  zurück, der durch `test2` selbst unverändert weitergegeben wird.

- h) Auf einem `Test`-Objekt wird die Methode `test1` aufgerufen. In `test1` wird ein Array, das Objekte des statischen Typs `A` speichert, erzeugt und zwei Objekte, eines vom Typ `A` und eines vom Typ `B`, darin gespeichert. Dann wird der gegebene Ausdruck ausgewertet:

- `ar[0].i` liefert den Wert 1, da Attribute statisch gebunden werden.
- `ar[0]` hat dynamisch den Typ A, somit wird die Methode `m` aus A ausgeführt, die  $1 * 100 = 100$  als Ergebnis von `ar[0].m()` liefert. (Siehe auch d))
- `ar[1]` hat zwar dynamisch den Typ B, statisch jedoch den Typ A. Somit wird das Attribut `i` der Klasse A benutzt. Das Ergebnis von `ar[1].i` ist also 1.
- `ar[1]` hat dynamisch den Typ B, also wird `m` aus B ausgeführt. Dies führt zu  $10 * 1000 = 10000$  als Ergebnis für `ar[1].m()`. (Siehe auch e))

An der mit `/* 5 */` gekennzeichneten Stelle wird also 10102 ausgegeben.

- i) Statisch hat die Variable `this` an der mit `/* 6 */` gekennzeichneten Stelle immer den Typ A. Dynamisch kann sie jedoch Objekte beliebiger Subtypen von A referenzieren. Diese könnten z.B. die Methode erben oder mit `super` aufrufen.



# Studierhinweise zur Kurseinheit 5

Diese Kurseinheit beschäftigt sich mit Kapitel 5 des Kurstextes. In ihrem Mittelpunkt steht eine allgemeine Einführung in Programmgerüste und die Behandlung des AWT, dem Gerüst zur Entwicklung von graphischen Bedienoberflächen in Java. Sie sollten den gesamten Text zu dieser Kurseinheit im Detail studieren und verstehen. Erweitern und modifizieren Sie die im Text angegebenen Beispiele, um mit den Komponenten und der Ereignissteuerung des AWT besser vertraut zu werden. Prüfen Sie nach dem Durcharbeiten des Kurstextes, ob Sie die Lernziele erreicht haben. Arbeiten Sie dazu auch wieder die Aufgaben am Ende der Kurseinheit durch.

## **Lernziele:**

- Programmgerüste: Was sind Gerüste? Wie unterscheiden sie sich von unabhängigen bzw. eigenständigen Bausteinen? Wofür werden sie benötigt?
- Aufgaben und Aufbau graphischer Bedienoberflächen.
- Struktur und Wirkungsweise des AWT; Zusammenspiel von Komponenten, Darstellung und Ereignissteuerung.
- Abstrakte Modelle als Grundlage für Programmgerüste: Was versteht man unter abstrakten Modellen? Wofür werden sie benötigt?
- Programmtechnische Kenntnisse zur Realisierung einfacher Bedienoberflächen.
- Zusammenhang zwischen Programmgerüsten und Software-Architekturen.
- Grundkonzept der Model-View-Controller-Architektur als Architekturbeispiel.

**Lernziele (Fortsetzung):**

- Entwicklung einer GUI ausgehend von einer Anforderungsbeschreibung; insbesondere: Entwurf der Dialogführung zur Anwendung, der Darstellung und der Steuerung.

In dieser Kurseinheit werden wir uns stärker als in den zurückliegenden mit programmtechnischen Aspekten beschäftigen. Dies hat drei Gründe:

- Erstens soll an einem realistischen Beispiel der Zusammenhang zwischen informellen Anforderungen, einem abstrakten, gedanklichen Modell und einer programmtechnischen Realisierung dieses Modells dargestellt werden.
- Zweitens dient uns das AWT dazu, ein typisches Anwendungsszenario vorzustellen, in dem viele Klassen benutzt werden, deren Eigenschaften man nicht in allen Details kennt.
- Drittens will die Kurseinheit bestimmte programmiertechnische Fertigkeiten vermitteln.

# Kapitel 5

## Objektorientierte Programmgerüste

Dieses Kapitel gibt zunächst eine kurze Einführung in objektorientierte Programmgerüste (Abschn. 5.1). Dann erläutert es am Beispiel des Abstract Window Toolkit – Javas Programmgerüst für die Entwicklung von Bedienoberflächen (kurz als AWT bezeichnet) –, wie Gerüste im Rahmen der objektorientierten Programmierung realisiert und eingesetzt werden können. Die Darstellung geht dabei vom abstrakten Modell aus, das dem AWT zugrunde liegt, und erläutert seine programmtechnische Umsetzung bis zu einem Detaillierungsgrad, der ausreicht, um kleinere Bedienoberflächen implementieren zu können (Abschn. 5.2). Danach wird kurz auf den methodischen Hintergrund zur Entwicklung graphischer Bedienoberflächen eingegangen. Ziel dabei ist es insbesondere, den engen Zusammenhang zwischen Software-Architekturen und Programmgerüsten an einem Beispiel kennen zu lernen (Abschn. 5.3).

Insgesamt verfolgt dieses Kapitel also mehrere Aspekte gleichzeitig: Einerseits sollen die Begriffe „Programmgerüst“, „abstraktes Modell“ und „Software-Architektur“ mit Leben erfüllt werden, indem sie anhand des Abstract Window Toolkit und dessen methodischen Hintergrunds exemplarisch erläutert werden. Andererseits bietet das Kapitel die programmtechnischen Grundlagen, die zur Realisierung graphischer Bedienoberflächen mit dem AWT nötig sind.

Das statt des AWT häufig verwendete Programmgerüst Swing baut auf dem AWT auf. Die Grundlagen von Swing sind daher dieselben wie die des AWT, insbesondere, was die Ereignisbehandlung angeht. Ein Umstieg auf das reichhaltigere Swing ist unproblematisch. An Swing Interessierte seien auf die Literatur verwiesen (z.B. [Krü07]) sowie auf die API-Dokumentation. Swing-Komponenten finden Sie im Paket `javax.swing`.



## 5.1 Programmgerüste: Eine kurze Einführung

Ein Ziel der Programmierung ist es, gute Lösungen wieder zu verwenden. Wiederverwendung kann sich dabei auf ganz unterschiedliche Abstraktionsebenen erstrecken. Die Spannbreite reicht von der Programmierung im Kleinen (Wiederverwendung einzelner Klassen bzw. Algorithmen) über die Programmierung im Großen (Wiederverwendung von Programmgerüsten und zusammenarbeitenden Modulen) bis zum Software-Entwurf (Wiederverwendung bestimmter architektonischer Muster). Auf allen Ebenen geht es darum, existierende, bewährte Lösungen von bestimmten softwaretechnischen Problemstellungen für ähnlich gelagerte Aufgabenstellungen zu nutzen und damit die Produktivität und Qualität der Programmierung zu steigern. Da es selten der Fall ist, dass eine existierende Lösung auch genau zur neuen Aufgabenstellung passt, ist Wiederverwendung meist mit Anpassung und Erweiterung verbunden.

Die objektorientierte Programmierung bietet auf allen Abstraktionsebenen Unterstützung für Wiederverwendung. Wie wir in den letzten beiden Kapiteln gesehen haben, kann man durch Subtyping und Vererbung einzelne Klassen erweitern und spezialisieren und sie damit an neue Anforderungen anpassen. Diese Techniken entfalten aber erst ihre volle Stärke, wenn man sie auf Systeme von Klassen anwendet. Unter einem *System von Klassen* verstehen wir dabei eine Ansammlung von abstrakten und konkreten Klassen und Schnittstellen, die zusammenwirken, um eine softwaretechnische Aufgabe zu lösen, die über die Funktionalität einer einzelnen Klasse hinausgeht.

In diesem Sinne bilden viele größere objektorientierte Anwendungsprogramme Systeme von Klassen, die sich allerdings meistens noch in Teilsysteme zerlegen lassen. Aus Sicht der Wiederverwendung ist es nun von großem Interesse, diejenigen Teilsysteme zu isolieren, die im Rahmen vieler Anwendungsprogramme genutzt werden können. Dazu müssen diese Teilsysteme von den anwendungsabhängigen Aspekten befreit und so entworfen werden, dass sie an die spezifischen Anforderungen verschiedener Anwendungsprogramme angepasst werden können. Wenn es gelingt, ein solches Teilsystem für viele unterschiedliche Anwendungen nutzbar zu machen, erreicht man nicht nur die Wiederverwendung von einzelnen, weitgehend unabhängigen Klassen, sondern auch die Wiederverwendung von Klassen, die in komplexer Weise zusammenwirken.

Ein *Programmgerüst* ist ein erweiterbares und anpassbares System von Klassen, das für einen allgemeinen, übergeordneten Aufgabenbereich eine Kernfunktionalität mit entsprechenden Bausteinen bereitstellt. Die Kernfunktionalität und die Bausteine müssen eine geeignete Grundlage bieten, um die Bewältigung unterschiedlicher konkreter Ausprägungen der allgemeinen Aufgabenstellung erheblich zu vereinfachen. Programmgerüste sind also Systeme von Klassen, die *speziell für* die Wiederverwendung bzw. Mehrfachver-

*System  
von Klassen*

*Programm-  
gerüst*

wendung entwickelt werden. Statt von Programmgerüsten werden wir im Folgenden abkürzend häufig nur von *Gerüsten* sprechen. Gleichbedeutend damit ist der englische Terminus *Framework*, der auch in deutschsprachiger Literatur oft benutzt wird<sup>1</sup>. Da der Terminus „Framework“ aber mit vielen anderen Bedeutungen überladen ist, verwenden wir die deutschsprachige Bezeichnung. Der Entwurf eines Gerüsts erfordert große softwaretechnische Erfahrung und eine genaue Kenntnis des Aufgabenbereichs bzw. der *fachlichen Domäne*, für den bzw. die das Gerüst bereitgestellt werden soll.

*Gerüst**Framework*

**Gerüst: Ja oder nein?** Die Erklärung des Begriffs „Programmgerüst“ ist nicht als scharfe Definition zu verstehen, sondern als wichtige Richtschnur, um Systeme von Klassen zu charakterisieren. Anhand von Beispielen wollen wir im Folgenden verdeutlichen, wann eine Menge von Klassen ein Programmgerüst ist und wann nicht. Wir betrachten zunächst Beispiele, die keine Gerüste sind, und skizzieren dann zwei Beispiele für Gerüste.

Im letzten Kapitel haben wir die Menge der Exception- und Error-Klassen behandelt. Sie bilden kein System von Klassen im obigen Sinne, also auch kein Gerüst. Sie wirken nicht zusammen, um eine Aufgabe zu erfüllen: Jedes Ausnahme-Objekt ist unabhängig von anderen Ausnahme-Objekten, kommuniziert nicht mit anderen Ausnahmen und erfüllt seine Aufgabe, das Transportieren von Fehlerinformation, selbständig.

Bei den Stromklassen ist man schon eher geneigt, sie als ein System von Klassen zu bezeichnen, insbesondere weil sie gut zusammenwirken können. Allerdings ist das Zusammenwirken nicht notwendig und nicht von vornherein auf die Lösung einer softwaretechnischen Aufgabe ausgerichtet. Stromklassen können auch völlig unabhängig voneinander eingesetzt werden. Deshalb bilden die Stromklassen kein System von Klassen.

Die Klassen im Paket `java.util` zur Verwaltung von Mengen, Listen und endlichen Abbildungen befinden sich schon an der Grenze zu einem System von Klassen. So wirken hier beispielsweise Iteratoren und Listen eng zusammen und erfüllen eine übergeordnete Aufgabe, die durch keine einzelne Klasse geleistet wird. Allerdings liegt die „übergeordnete“ Aufgabe noch sehr dicht an der Funktionalität einer einzelnen Klasse.

Als nächstes betrachten wir den Fall, in dem Klassen zwar Systeme bilden, aber trotzdem keine Programmgerüste sind. Beispielsweise ist eine fertige Anwendung, die durch ein System eng kooperierender Klassen realisiert ist, in der Regel kein Programmgerüst. Die Forderung, dass das System erweiterbar und anpassbar sein soll, um als Kernfunktionalität für unterschiedliche, anwendungsspezifische Software zu dienen, ist in diesem Fall im Allgemeinen nicht erfüllt. Insbesondere folgt nämlich aus der Anpassbarkeit und

<sup>1</sup>Wir folgen damit im Wesentlichen der deutschen Übersetzung des Begriffs „Framework“ von Hanspeter Mössenböck in [M98]; allerdings verwendet er Programmgerüst und Gerüst nicht synonym.

Erweiterbarkeit einzelner Klassen nicht, dass ein System von Klassen in angemessener Weise anpassbar und erweiterbar ist.

Schließlich skizzieren wir typische Programmgerüste. Dazu betrachten wir zwei allgemeine, übergeordnete Aufgabenbereiche: die Entwicklung graphischer Bedienoberflächen und die Erstellung der informationstechnischen Unterstützung zur Abwicklung von Geschäftsprozessen für ein Unternehmen.

1. Ein Gerüst für die Entwicklung graphischer Bedienoberflächen werden wir in den folgenden Abschnitten dieses Kapitels genauer kennen lernen. Es besteht aus einer Vielzahl von Klassen, die eng zusammenwirken, um Oberflächenkomponenten und die Steuerung der zugrunde liegenden Anwendung zu realisieren. Die beteiligten Klassen beschreiben nicht eine Oberfläche, sondern sie stellen eine Kernfunktionalität zur Verfügung, um Bedienoberflächen für beliebige Anwendungen zu implementieren.
2. Das von IBM entwickelte Programmgerüst „San Francisco“ stellt die Basisfunktionalität und entsprechende, anpassbare Bausteine bereit, mit denen man für die informationstechnischen Aufgaben in Unternehmen (Buchführung, Lagerhaltung, Auftragsabwicklung, interne Kommunikation, etc.) unternehmensspezifische, integrierte Lösungen realisieren kann (siehe z.B. [BNS99]). Das Programmgerüst bietet also wiederum keine fertige Lösung, sondern nur den Rahmen und wesentliche Bausteine, um eine spezifische Anwendung bzw. eine Menge kooperierender spezifischer Anwendungen zu entwickeln.

**Unvollständigkeit und Modellierung von Gerüsten.** Ähnlich wie eine abstrakte Klasse oder eine Schnittstelle bestimmte gemeinsame Eigenschaften aller ihrer Unterklassen und Subtypen zusammenfasst, stellt ein Gerüst nur die Kernfunktionalität für eine Aufgabe bereit und abstrahiert<sup>2</sup> von allen Aspekten einer speziellen Anwendung. Dies kann wie bei abstrakten Klassen dazu führen, dass Gerüste unvollständige Programmteile enthalten, die ohne entsprechende Erweiterung alleine nicht lauffähig sind. Das soll auch durch den Terminus Gerüst angedeutet werden: Das Gerüst fungiert als Rahmen und Skelett eines Systems, muss aber ggf. ausgefüllt und angereichert werden, um funktionsfähig zu werden.

Das Zusammenwirken der Klassen bzw. ihrer Objekte in einem Programmgerüst ist häufig recht komplex. Deshalb kann von den Anwendern eines Programmgerüsts im Allgemeinen nicht erwartet werden, dass sie dieses Zusammenwirken im Detail verstehen. Um ihnen dennoch die Nutzung auch komplexer Gerüste zu ermöglichen, sollte die Wirkungsweise eines

---

<sup>2</sup>Zum Begriff „Abstraktion“ vgl. Abschn. 3.1, S. 150.

Gerüsts in Form eines abstrakten Modells beschrieben sein. Das *abstrakte Modell* muss alle für den Anwender notwendigen Aspekte des Gerüsts erklären können, kann aber die Details der Implementierung und Systemanbindung vollständig oder zumindest weitgehend verbergen. In der objektorientierten Programmierung besteht ein abstraktes Modell aus den relevanten Objekten und der Art, wie sie miteinander kommunizieren.

*abstraktes  
Modell*

Darüber hinaus ist es sehr hilfreich darzulegen, welche architektonischen und methodischen Ideen hinter einem Gerüst stehen und wie sie zu nutzen sind, um Anwendungen mit dem Gerüst zu entwickeln. Das Zusammenspiel zwischen Software-Architektur, Programmgerüst und Entwicklungsmethodik illustrieren wir in Abschn. 5.3 an einem Beispiel.

## 5.2 Ein Gerüst für Bedienoberflächen: Das AWT

Dieser Abschnitt behandelt den Entwurf und die Anwendung von Programmgerüsten am Beispiel des AWT. Er bietet zunächst eine kurze Einführung in die Aufgaben und den Aufbau graphischer Bedienoberflächen. Dann erläutert er das abstrakte Modell, das dem AWT zugrunde liegt. Anschließend beschäftigt er sich detaillierter mit einigen Klassen und Mechanismen des AWT und illustriert deren Anwendung anhand kleinerer Beispiele.

### 5.2.1 Aufgaben und Aufbau graphischer Bedienoberflächen

Graphische Bedienoberflächen ermöglichen eine flexible, benutzerfreundliche Interaktion zwischen Benutzern und Anwendungsprogrammen, sogenannten *Anwendungen*. Graphische Bedienoberflächen werden im Englischen als graphical user interfaces oder kurz *GUIs* bezeichnet; wir werden diesen Begriff auch im Kurstext verwenden.

*Anwendung*

Drei Aspekte stehen bei der Interaktion mit einer Bedienoberfläche im Mittelpunkt:

1. Die *Steuerung* der Anwendung durch die GUI; technisch gesehen bedeutet das meist den Aufruf von Prozeduren/Methoden, die von der Anwendung bereitgestellt werden.
2. Die Eingabe oder Auswahl von Daten durch die Benutzer.
3. Die *Darstellung* des Zustands der Anwendung bzw. bestimmter Rechenergebnisse auf dem Bildschirm.

*Steuerung*

*Darstellung*

Eine GUI stellt sich einem Benutzer als eine Menge von Eingabe-, Bedien- und Darstellungselementen dar. Als Beispiel betrachten wir eine GUI für den in Kap. 3 entwickelten Browser (siehe Abb. 5.1). Durch Anklicken der Schaltflächen „zurueck“, etc. kann der Benutzer die Browser-Kommandos

ausführen. Mit Hilfe des Textfensters kann er Adressen für Seiten eingeben. Der untere Teil der GUI dient als Ausgabefenster.

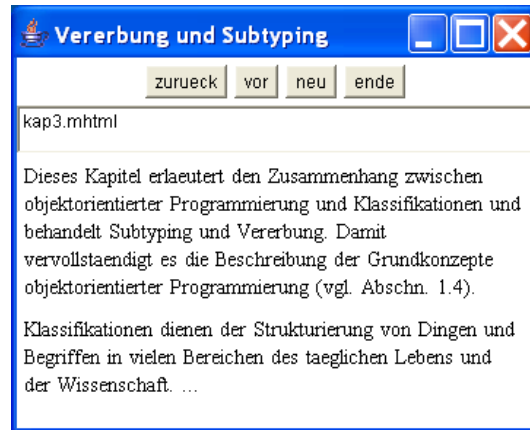


Abbildung 5.1: Bedienoberfläche eines einfachen Browser

Üblicherweise besitzen Oberflächen weitere Bedienkomponenten, beispielsweise Rollbalken (Scrollbars) und Auswahlmenüs. Ein Benutzer kann zu einem Zeitpunkt im Allgemeinen also unterschiedliche Aktionen ausführen, insbesondere an verschiedenen Stellen Text eingeben und zwischen mehreren Schaltflächen auswählen. Wir sagen, dass er viele Möglichkeiten hat, den *Dialog* mit der Anwendung fortzusetzen. Daraus resultiert eine große Anzahl potentieller Dialogabläufe. Auch muss der Benutzer nicht das Ende einer Operation abwarten, bis er weitere Operationen auslösen kann; d.h. er kann mehrere Operationen quasi gleichzeitig anstoßen (beispielsweise kann er kurz nacheinander auf zwei Schaltflächen klicken, so dass die Aktion, die er mit der ersten Schaltfläche aufgerufen hat, beim Anklicken der zweiten noch nicht beendet ist). Diese flexible Steuerung von Anwendungen bringt drei Schwierigkeiten mit sich:

1. Zulässigkeit der angestoßenen Operationen: Während der Bedienung einer Anwendung sind nicht in jedem Zustand alle Operationen zulässig. Die Bedienoberfläche muss dafür sorgen, dass nur die aktuell zulässigen Operationen vom Benutzer ausgelöst werden können. Im Übrigen muss sichergestellt werden, dass alle Parameter einer Operation eingegeben sind, bevor sie ausgelöst wird.
2. Konsistenz der Darstellung: Die graphische Darstellung der Anwendung auf dem Bildschirm hängt von deren Zustand ab. Ändert sich der Zustand (etwa durch Ausführung einer Operation), müssen alle Teile der Darstellung konsistent nachgeführt werden.
3. Nebenläufigkeit: Um die oben skizzierte Flexibilität zu gewährleisten, müssen Modelle für Bedienoberflächen zumindest konzeptionell die

Möglichkeit vorsehen, dass mehrere Operationen quasi parallel ausgelöst werden können. Allerdings vergrößert diese Möglichkeit die ersten beiden Schwierigkeiten erheblich<sup>3</sup>.

Um die Entwicklung und Implementierung graphischer Bedienoberflächen zu erleichtern, gibt es mittlerweile diverse Programmgerüste, die oft auch als Oberflächen-*Toolkits* bezeichnet werden. Diese Gerüste stellen für alle wesentlichen Aufgaben der Oberflächenentwicklung Grundbausteine zur Verfügung und bieten die Standardfunktionalität zur Darstellung der Oberflächenkomponenten auf dem Bildschirm und zur Behandlung und Weiterleitung der Benutzereingaben. Wie derartige Gerüste üblicherweise aufgebaut sind, werden wir im nächsten Abschnitt am Beispiel des AWT erläutern.

*Toolkit*

### 5.2.2 Die Struktur des Abstract Window Toolkit

Das Abstract Window Toolkit ist ein hervorragendes Beispiel, um Programmgerüste zu studieren. Programmgerüste für GUIs sollen den Oberflächenentwicklern ein möglichst einfaches *abstraktes Modell* graphischer Bedienoberflächen zur Verfügung stellen. Dieses abstrakte GUI-Modell soll die systemnahen Aspekte der zugrunde liegenden Fenstersysteme so einkapseln, dass die Entwickler diese häufig recht komplexen Details nicht kennen müssen. Sie brauchen nur die Klassen und Methoden des AWT zu kennen und zu wissen, wie diese im Rahmen des abstrakten Modells arbeiten. Beim AWT kam zusätzlich die Forderung hinzu, dass das bereitgestellte abstrakte GUI-Modell von den zugrunde liegenden Fenstersystemen weitgehend unabhängig sein sollte, sodass eine mit dem AWT erstellte GUI auf vielen unterschiedlichen Fenster- und Betriebssystemen laufen kann (insbesondere natürlich unter X-Windows/Unix und unter Microsoft Windows).

*abstraktes  
Modell*

Das AWT umfasst eine große Anzahl von Klassen, die zusammen ein relativ komplexes Gerüst bilden, um GUIs zu realisieren. Die Komplexität rührt zum einen daher, dass viele Klassen gegenseitig direkt oder indirekt voneinander abhängen, d.h. verschränkt rekursiv sind (Unterabschn. 5.2.2.5 skizziert dazu ein Beispiel). Zum anderen liegt dem Gerüst auch ein recht enges dynamisches Zusammenwirken der einzelnen Teile zugrunde. Dieser Abschnitt führt in das abstrakte Modell ein, das durch dieses Zusammenwirken realisiert wird und damit die konzeptionelle Schnittstelle zum Oberflächenentwickler bildet. Der Schwerpunkt liegt dabei auf der Darstellung der zentralen Konzepte und darauf, wie diese Konzepte mittels objektorientierter Techniken im Rahmen des AWT umgesetzt sind.

---

<sup>3</sup>Beispiel: In einem Zustand einer Anwendung können die Operationen A und B ausgelöst werden. Die Durchführung von A führt zu einem Zustand, in dem B nicht mehr zulässig ist. Was soll passieren, wenn B direkt nach A angestoßen wird, d.h. bevor A die Zustandsänderung bewirkt und das Auslösen von B damit ausgeschlossen hat?

Wir beschreiben zunächst das abstrakte Modell fürs AWT. Danach erläutern wir etwas genauer dessen Komponenten, wichtige Aspekte der graphischen Darstellung und die Ereignissteuerung. Schließlich geben wir eine kurze Übersicht über die programmtechnische Realisierung des AWT mittels einer Menge von Klassen.

### 5.2.2.1 Das abstrakte GUI-Modell des AWT

Abschnitt 5.2.1 hat bereits alle wesentlichen Aufgaben angedeutet, die GUIs bewältigen müssen. Texte, Graphiken, Bilder und Bedienelemente, wie Schaltflächen, Menüs, Rollbalken u.Ä., sind am Bildschirm zu platzieren und anzuzeigen. Dazu werden insbesondere Techniken benötigt, mit Hilfe derer die Platzierung dieser *Darstellungselemente* in einem Fenster so beschrieben werden kann, dass sich nach Vergrößern bzw. Verkleinern eines Fensters wieder eine akzeptable Darstellung ergibt. Beispielsweise müssen die Schaltflächen der Bedienoberfläche von Abb. 5.1 nach einer Größenveränderung neu zentriert werden. Des Weiteren muss ein Mechanismus existieren, der verdeckte Fensterinhalte restauriert, wenn das Fenster am Bildschirm wieder in den Vordergrund tritt. Zu den Aufgaben der Steuerung gehört es, Benutzereingaben (Mausbewegungen, -klicks, Tastatureingaben) den Fenstern und Bedienelementen zuzuordnen und die gewünschten Reaktionen zu veranlassen.

Es gibt prinzipiell eine Reihe recht unterschiedlicher Möglichkeiten, die skizzierten Aufgaben durch ein Programmgerüst zu unterstützen. Zum Teil hängen die Lösungsmöglichkeiten auch von der verwendeten Implementierungssprache bzw. dem zugrunde liegenden Programmierparadigma ab. Für uns steht hier die Nutzung objektorientierter Techniken im Vordergrund, insbesondere natürlich der Einsatz von Vererbung und die Kommunikation über Nachrichten. Drei Aspekte spielen dabei in dem vom AWT bereitgestellten abstrakten Modell eine zentrale Rolle:

1. **Komponenten:** Jedem Darstellungselement einer GUI entspricht im AWT ein Objekt vom Typ `Component`. Diese Objekte werden wir im Folgenden *Oberflächenkomponenten* oder kurz *Komponenten* nennen.
2. **Darstellung:** Es muss beschrieben werden, wie die einzelnen Komponenten auf dem Bildschirm dargestellt werden und wie die Darstellungen der Komponenten angeordnet werden sollen.
3. **Ereignissteuerung:** Das Fenstersystem benachrichtigt die Komponenten über Benutzereingaben und Veränderungen auf der Fensteroberfläche. Das Eintreffen derartiger Nachrichten über Benutzereingaben und Veränderungen bezeichnet man als ein *Ereignis* (engl. *Event*). Oft werden auch die Nachrichten selbst Ereignisse genannt. Die Komponenten leiten diese Nachrichten, ggf. in modifizierter Form, an sogenannte Beobachter-Objekte weiter, die die Steuerung der GUI übernehmen.

*Oberflächenkomponente*

*Ereignis*

Bevor wir uns diese drei Aspekte im Einzelnen anschauen, skizzieren wir denjenigen Teil des abstrakten GUI-Modells, der von Java unabhängig ist und demnach auch für Fensteroberflächen gilt, die nicht mit dem AWT erstellt wurden.

Komponenten können im Allgemeinen hierarchisch ineinander geschachtelt sein, d.h. es gibt Komponenten, die eine oder mehrere Komponenten enthalten, die ihrerseits selbst wieder Komponenten enthalten (z.B. enthält das Bedienfenster von Abb. 5.1 unterhalb der Titelleiste eine Komponente, die selbst wieder mehrere Schaltflächen enthält). Es gibt Komponenten, die in keiner anderen enthalten sein können. Diese „Top-Level“-Komponenten, die in der Schachtelungshierarchie am weitesten oben stehen, heißen *Fenster*. Ein Fenster ist entweder ikonifiziert oder nicht ikonifiziert, es ist entweder *aktiv* oder *inaktiv*. Das Fenstersystem garantiert, dass immer nur eines der Fenster auf dem Bildschirm aktiv ist, alle anderen sind inaktiv. Eine Komponente im aktiven Fenster oder das aktive Fenster selbst besitzt den sogenannten *Fokus*. Die Komponente mit dem Fokus erhält die Tastatureingaben. Außerdem gibt es zu jedem Zeitpunkt genau eine *Mausposition* (die Mausposition muss nicht im aktiven Fenster liegen). Diese allgemeinen Grundlagen sind vor allem für das Verständnis der Ereignissteuerung wichtig.

*aktiv/inaktiv**Fokus*

### 5.2.2.2 Komponenten

Komponenten sind die Grundbausteine, aus denen GUIs aufgebaut werden. Im Folgenden geben wir einen Überblick über die Komponententypen des AWT. In Abschn. 5.2.3 werden wir einige dieser Komponenten detaillierter behandeln. Das AWT unterscheidet zwei Arten von Komponenten: *elementare Komponenten* und *Behälter-Komponenten*. Elementare Komponenten enthalten keine anderen Komponenten. Das AWT stellt u.a. die folgenden elementaren Komponenten zur Verfügung:

*elementare  
bzw. Behälter-  
Komponente*

- Canvas/Zeichenfläche: Sie dienen der Darstellung von gezeichneten Graphiken oder Texten.
- Label: Sie dienen zur Darstellung einer Zeichenreihe auf der GUI.
- Buttons/Schaltflächen: Schaltflächen empfangen Mausklicks und leiten sie an die Steuerung weiter.
- Auswahlkomponenten: Das AWT stellt unterschiedliche Komponenten bereit, mit denen der Benutzer aus einer Liste von Einträgen auswählen kann, z.B. durch Anklicken kleiner Kästchen (CheckBox-Komponenten) oder durch Auswahl aus einer Liste (bei List-Komponenten sind die Listenelemente immer sichtbar, bei Choice-Komponenten klappt die Liste bei Mausberührung auf).



- Scrollbars/Rollbalken: Mit ihnen kann man Komponenten scrollbar machen. So können z.B. Texte oder Grafiken, die nicht vollständig auf dem Bildschirm angezeigt werden können, durch Scrollen sichtbar gemacht werden.
- Textkomponenten: Sie dienen der Eingabe und dem Anzeigen von Text.

Behälter-Komponenten werden im Wesentlichen eingesetzt, um andere Komponenten zu gruppieren und anzuordnen. Dafür besitzen sie die Methoden `add` zum Hinzufügen und `remove` zum Entfernen von Komponenten.

Die einfachsten Behälter, die das AWT zur Verfügung stellt, sind Objekte der Klasse `Panel`. Darüber hinaus gibt es Behälter, die das Scrollen ihres Inhalts unterstützen (siehe die Klasse `ScrollPane`), und Fenster. Wie oben bereits beschrieben, sind *Fenster* Behälter, die ganz außen liegen, d.h. insbesondere zu keinem anderen Behälter hinzugefügt werden können. (Diese Eigenschaft, die Fenster von allen anderen Komponenten unterscheidet, wird zur Laufzeit überprüft; beim Versuch, ein Fenster einem Behälter hinzuzufügen, wird eine Ausnahmebehandlung eingeleitet.) Das AWT kennt zwei Arten von Fenstern:

*Haupt-  
fenster*

1. *Hauptfenster*, das sind Objekte der Klasse `Frame`; ein Hauptfenster ist keinem anderen Fenster zugeordnet und bildet quasi den zentralen Aufhänger für zusammenhängende Oberflächenteile.

*Dialog-  
fenster*

2. *Dialogfenster*, das sind Objekte der Klasse `Dialog`; ein Dialogfenster ist immer einem anderen Fenster zugeordnet und kann die Aktivitäten in diesem Fenster und allen darin enthaltenen Komponenten blockieren, solange der Dialog mit dem Benutzer läuft.

Hauptfenster spielen eine wichtige Rolle beim Zusammenspiel mit dem zugrunde liegenden Fenstersystem. Komponenten, die keine Fenster sind, werden nämlich nur dann auf dem Bildschirm gezeigt, wenn sie in einem Fenster enthalten sind. Und Fenster, die keine Hauptfenster sind, werden nur dann gezeigt, wenn sie einem Hauptfenster zugeordnet wurden. Die gemeinsamen Eigenschaften von Fenstern sind in der Klasse `Window` zusammengefasst, die Gemeinsamkeiten von Behältern in der Klasse `Container` und die Gemeinsamkeiten aller Komponenten in der Klasse `Component`. Eine vollständigere Klassifikation der AWT-Komponenten bietet Abbildung 5.2. Dabei wurden direkt die Klassennamen des AWT verwendet. `Component` ist eine abstrakte Klasse. Alle genannten Klassen gehören zum Paket `java.awt`; nur die Klasse `Applet` gehört zum Paket `java.applet`.

Komponenten besitzen u.a. eine Hintergrund- und Vordergrundfarbe, eine bevorzugte, minimale, maximale und aktuelle Größe und eine Position. Die Position bezieht sich bei Fenstern auf die Koordinaten des Bildschirms,

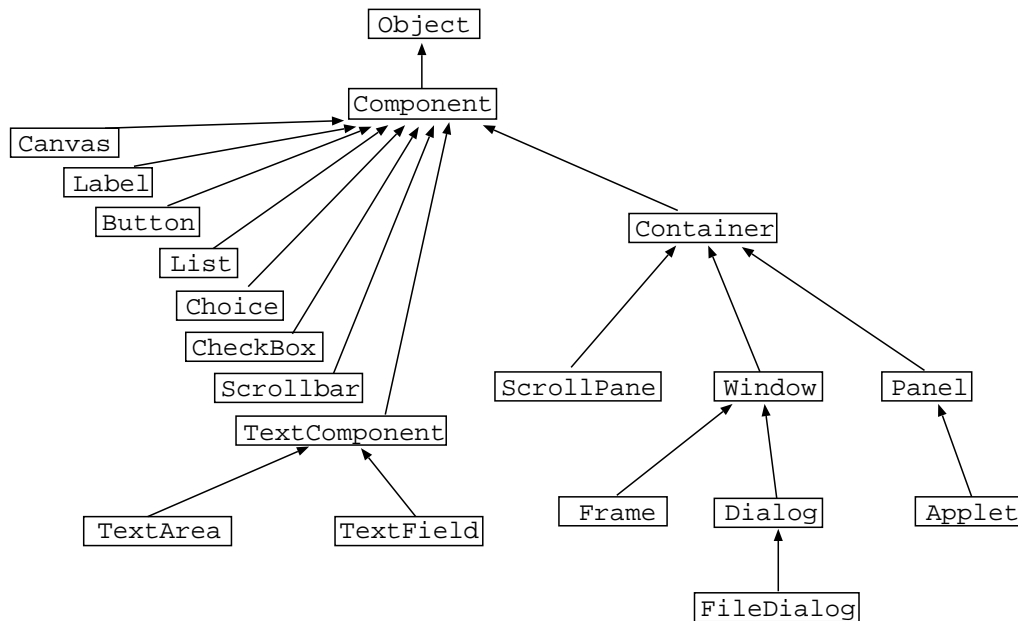


Abbildung 5.2: Die Komponentenklassen des AWT

bei allen anderen Komponenten auf die der umfassenden Behälter-Komponente. Sie können auf dem Bildschirm *sichtbar* sein (engl. *visible*); sie können *aktiv* sein (engl. *enabled*), d.h. bereit sein, mit dem Benutzer zu interagieren; sie können eine *gültige* Darstellung besitzen (engl. *valid*); oder sie sind nicht sichtbar bzw. inaktiv oder ihre Darstellung ist momentan ungültig (nicht aktuell), d.h. sie müsste neu gezeichnet werden z.B. nach Änderung ihrer Größe oder weil sie zuvor von einem anderen Fenster überdeckt wurde.

Die Klasse `Component` besitzt eine Vielzahl öffentlicher Methoden, um diese und weitere Attribute zu manipulieren und abzufragen sowie um Beobachter-Objekte für die Ereignissteuerung anzubinden (siehe unten). Alle diese Methoden werden selbstverständlich an die Subklassen vererbt – hier kann man also einmal an einem realistischen Beispiel sehen, was Vererbung in der Praxis bedeuten kann.

### 5.2.2.3 Darstellung

Jeder sichtbaren Komponente ist ein rechteckiges Darstellungselement auf dem Bildschirm zugeordnet, das es graphisch repräsentiert. Für jeden Komponententyp des AWT ist eine bestimmte graphische Darstellung voreingestellt. Der GUI-Entwickler kann aber einzelne Parameter, wie den Font oder die Hintergrundfarbe, verändern. Wie wir sehen werden, kann er insbesondere das Layout-Verfahren bestimmen, mit dem die Darstellungselemente, die zu Komponenten eines Behälters gehören, relativ zueinander angeordnet werden sollen. Abbildung 5.3 zeigt den Zusammenhang zwischen

Komponenten und den zugehörigen Darstellungselementen am Beispiel der Browser-Oberfläche.

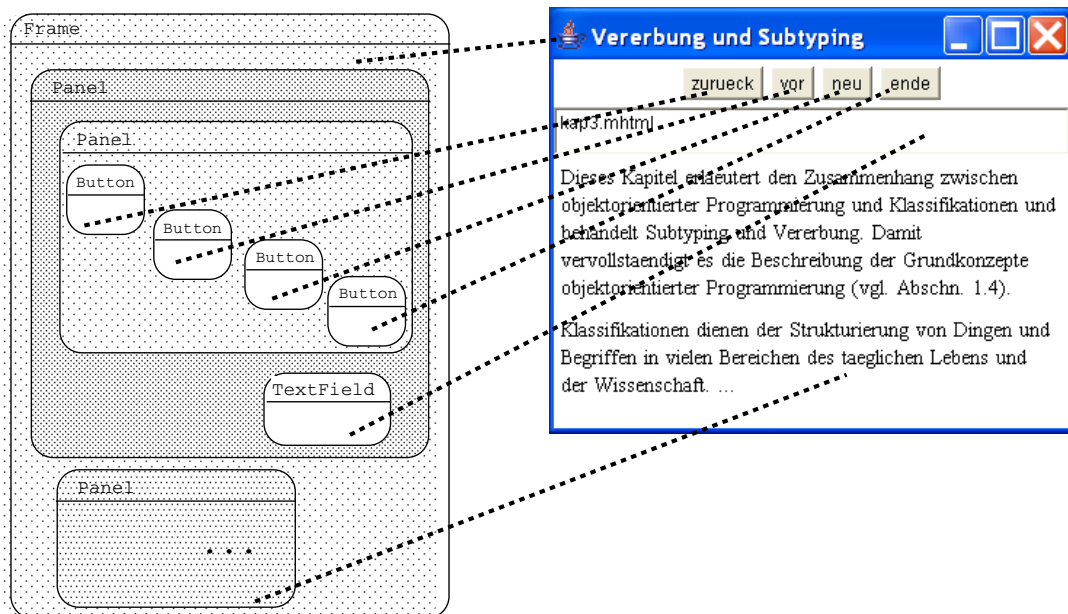


Abbildung 5.3: Zuordnung von Komponenten zu Darstellungselementen

Die linke Hälfte der Abbildung veranschaulicht die Komponenten. Behälter-Objekte sind dabei so gezeichnet, dass sie die in ihnen enthaltenen Komponenten umfassen. Beispielsweise enthält die Frame-Komponente zwei Panel-Objekte. Bereits aus dieser vereinfachten Skizze kann man erkennen, dass Panel üblicherweise keine entsprechenden Darstellungselemente besitzen, sondern hauptsächlich der Anordnung der in ihnen enthaltenen Objekte dienen.

**Darstellen einzelner Komponenten.** Sichtbare Komponenten besitzen eine graphische Darstellung auf dem Bildschirm. Die Zustandsinformation zu der graphischen Darstellung wird programmtechnisch durch ein Objekt der Klasse `Graphics` repräsentiert (die Methode `getGraphics` liefert das `Graphics`-Objekt einer Komponente). Sie umfasst u.a. die aktuelle Farbe und den aktuellen Font. `Graphics`-Objekte besitzen darüber hinaus eine Reihe von Methoden zum Zeichnen. Die Zeichenoperationen, die von dem `Graphics`-Objekt einer sichtbaren Komponente ausgeführt werden, werden direkt auf dem Bildschirm wiedergegeben.

Da im `Graphics`-Objekt nicht festgehalten wird, was gezeichnet wurde, muss die graphische Darstellung einer Komponente jedes Mal neu gezeichnet werden, wenn z.B. eine bislang verdeckte Komponente auf dem Bildschirm wieder in den Vordergrund kommt. Damit der Oberflächenprogrammierer

sich um dieses Neu-Zeichnen nicht kümmern muss, hat das AWT dafür einen Mechanismus vorgesehen: Jede Komponente besitzt eine Methode `paint`. Wenn das Fenstersystem feststellt, dass ein (Neu-)Zeichnen der Darstellung einer Komponente erforderlich ist, schickt es dieser Komponente eine Nachricht, die automatisch den Aufruf von `paint` veranlasst. Durch Überschreiben von `paint` ist es also möglich, das Aussehen von Komponenten dauerhaft zu verändern.

Will ein Programmierer das Neu-Zeichnen einer Komponente explizit auslösen, z.B. weil er das Aussehen der Komponente verändert hat, kann er dies durch Aufruf der Methode `repaint` erreichen. Durch diesen Aufruf wird (ggf. über den Umweg des Aufrufs der Methode `update`) der Aufruf der Methode `paint` ausgelöst<sup>4</sup>.

**Darstellen von Behältern.** Die graphische Darstellung einer Behälter-Komponente  $B$  wird aus den Darstellungen der im Behälter enthaltenen Komponenten  $E_1, \dots, E_n$  berechnet. Dies ist im Allgemeinen eine nicht-triviale Aufgabe. Wie sollen die  $E_i$ 's im Rahmen der Darstellung von  $B$  platziert werden? Der erste Ansatz wäre, ihnen beim Einfügen in den Behälter feste Koordinaten zuzuordnen und über die Reihenfolge des Einfügens festzulegen, welche Komponenten im Vordergrund liegen, wenn sich ihre Darstellungen überlappen. Dieser Ansatz hat aber viele Nachteile. Insbesondere wird bei einer solchen Lösung i.A. die Darstellung nicht angepasst, wenn sich beispielsweise die Größe des äußersten Fensters verändert.

Deshalb bietet das AWT sogenannte *Layout-Manager* an, die automatisch die Darstellung eines Behälters  $B$  aus den Darstellungen der  $E_i$ 's berechnen. Da es kein Layout-Berechnungsverfahren gibt, das allen Wünschen genügt, stellt das AWT mehrere unterschiedliche Layout-Manager zur Verfügung und ermöglicht es darüber hinaus, weitere Layout-Manager selbst zu implementieren (meist durch Spezialisierung von einem der vordefinierten Layout-Manager). Jeder Behälter besitzt eine Referenz auf einen Layout-Manager, den man mit den Methoden `getLayout` und `setLayout` abfragen und neu einstellen kann.

*Layout-  
Manager*

Wird eine Komponente zu einem Behälter  $b$  hinzugefügt oder aus ihm entfernt, kann der Programmierer durch Aufruf der Methoden-Sequenz `b.invalidate(); b.validate()` erreichen, dass der zugehörige Layout-Manager die Darstellung neu berechnet und die geänderte Darstellung angezeigt wird.

---

<sup>4</sup>Nähere Informationen zu diesem Thema sind zu finden unter der URL "<http://java.sun.com/products/jfc/tsc/articles/painting/index.html>". (Letzter Zugriff: 10.03.2007.)

#### 5.2.2.4 Ereignissteuerung

Bisher haben wir nur betrachtet, durch welche Objekte eine graphische Bedienoberfläche repräsentiert und dargestellt wird. Dieser Unterabschnitt erläutert, wie das AWT Aspekte der Steuerung unterstützt, d.h. welche Mechanismen es bereitstellt, um auf Ereignisse wie Veränderungen der Oberfläche und Eingaben durch Benutzer zu reagieren. Typische Ereignisse sind z.B. Mausklicks, Mausbewegungen, Tastatureingaben oder das Überdecken eines Fensters durch ein anderes Fenster. Wir geben zunächst einen Überblick über die im AWT vorgesehenen Ereignisse und deren Klassifikation. Dann beschreiben wir, wie auftretende Ereignisse zur Steuerung herangezogen werden können. Bestimmte Ereignisse können nämlich durch geeignete Objekte beobachtet werden, die dann entsprechend der ihnen gemeldeten Ereignisse reagieren können.

Ereignis  
tritt an  
Komponente  
auf

**Ereignisse.** Jedes auftretende Ereignis wird vom Fenstersystem einer Komponente zugeordnet. Beispielsweise wird ein Mausklick der Komponente zugeordnet, in deren sichtbarer, nicht verdeckter Darstellung die Mausposition liegt. Eine Tastatureingabe wird der Komponente mit dem Fokus (siehe S. 309) zugeordnet. Eine Veränderung eines Fensters wird diesem Fenster zugeordnet. Anstatt zu sagen, das auftretende Ereignis  $E$  wird der Komponente  $k$  zugeordnet, sagen wir abkürzend, das Ereignis  $E$  tritt an der Komponente  $k$  auf. Die Komponente  $k$  wird häufig auch die *Quelle* von  $E$  genannt.

elementare  
u. semantische  
Ereignisse

Das AWT unterscheidet zwischen *elementaren* Ereignissen (engl. *Low-Level Events*) und *semantischen* Ereignissen (engl. *Semantic Events*). Die meisten elementaren Ereignisse können an jeder Komponente auftreten, zwei nur an Behälter-Komponenten und einige nur an Fenstern. Die folgende Aufstellung gibt einen Überblick über die elementaren Ereignisse. Wir benutzen die englischen Namen des AWT für die Ereignisse und fassen sie zu *Ereignissorten* zusammen. Die genannten Ereignissorten entsprechen Klassen aus dem Paket `java.awt.event`. Im Folgenden werden wir die Ereignisse, die zu den verschiedenen Ereignissorten gehören, auflisten. Die Bedeutung der Ereignisse sollte intuitiv klar sein (einige werden in einer Fußnote erläutert). Die möglichen Ereignisse sind in den von `AWTEvent` abgeleiteten Klassen in Form von Konstanten (IDs) festgehalten. Da auftretende Ereignisse aber, wie wir noch sehen werden, nur über ihnen eineindeutig zugeordnete Methoden kommuniziert werden, identifizieren wir die Ereignisse im Folgenden über die Namen dieser Methoden und nicht über die Konstanten.

- An Komponenten treten Ereignisse der Sorten `ComponentEvent`, `FocusEvent`, `KeyEvent` und `MouseEvent` auf und zwar:  
*ComponentEvents*: `componentMoved`, `componentHidden`, `componentResized`, `componentShown`;  
*FocusEvents*: `focusGained`, `focusLost`;

*KeyEvents*: keyPressed, keyReleased, keyTyped<sup>5</sup>;

*MouseEvents*: mousePressed, mouseReleased, mouseClicked<sup>6</sup>, mouseEntered, mouseExited, mouseDragged, mouseMoved.

- An Behältern treten zusätzlich Ereignisse der Sorte ContainerEvent auf:  
*ContainerEvents*: componentAdded, componentRemoved.
- An Fenstern treten zusätzlich Ereignisse der Sorte WindowEvent auf:  
*WindowEvents*: windowOpened, windowClosing<sup>7</sup>, windowClosed, windowIconified, windowDeiconified, windowActivated, windowDeactivated, windowGainedFocus, WindowLostFocus, WindowStateChanged.

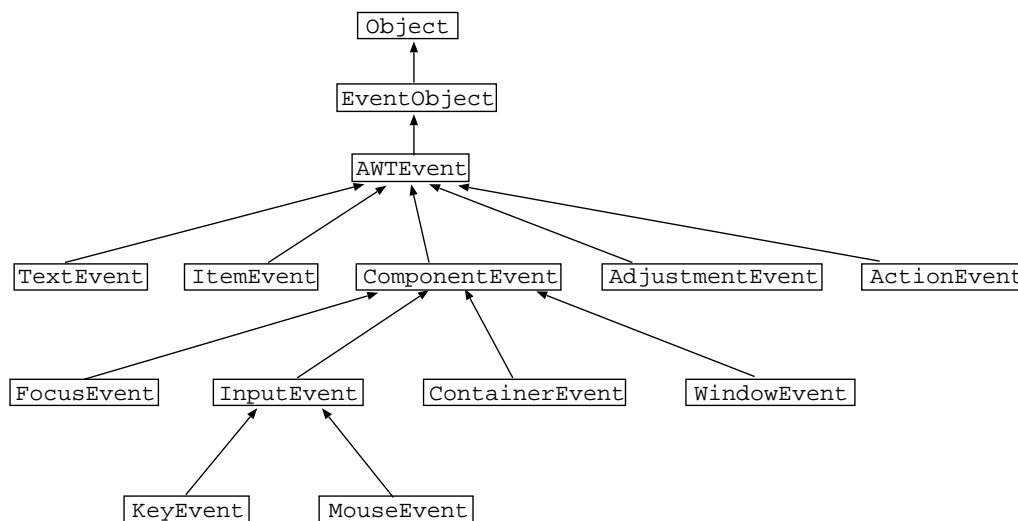


Abbildung 5.4: Die Hierarchie der Event-Klassen des AWT

Semantische Ereignisse fassen mehrere elementare Ereignisse zu einem Ereignis zusammen, um das Programmieren der Ereignissteuerung zu vereinfachen. Tritt zum Beispiel an einer Schaltfläche ein mousePressed-Ereignis für die linke Maustaste gefolgt von einem entsprechenden mouseReleased-Ereignis auf, ohne dass die Mausposition zwischenzeitlich die Komponente verlassen hat, wird zusätzlich an dieser Schaltfläche ein actionPerformed-Ereignis erzeugt. Will man mit einer Schaltfläche eine Operation steuern, reicht

<sup>5</sup>Der Kommentar im AWT sagt: „The ‘key typed’ event. This event is generated when a character is entered. In the simplest case, it is produced by a single key press. Often, however, characters are produced by series of key presses, and the mapping from key pressed events to key typed events may be many-to-one or many-to-many.“

<sup>6</sup>Ein Mausklick ereignet sich, wenn das Drücken und Loslassen einer Maustaste an der gleichen Mausposition erfolgt.

<sup>7</sup>Das Ereignis „windowClosing“ wird erzeugt, wenn der Benutzer die Anweisung zum Schließen eines Fensters gegeben hat.

es somit aus, auf `actionPerformed`-Ereignisse zu reagieren. Außer an Komponenten vom Typ `Button` treten `actionPerformed`-Ereignisse an Komponenten vom Typ `List`, `MenuItem` und `TextField` auf. Das AWT kennt drei weitere semantische Ereignisse: `adjustmentValueChanged` für Rollbalken, `itemStateChanged` für Auswahlkomponenten und `textValueChanged` für Textkomponenten. Für jedes semantische Ereignis gibt es eine Ereignissorte, die genau dieses Ereignis enthält; die Sorten heißen `ActionEvent`, `AdjustmentEvent`, `ItemEvent` und `TextEvent`. Um die Sprechweise zu vereinfachen sagen wir, dass *einer Komponente eine Ereignissorte zugeordnet ist*, wenn Ereignisse dieser Sorte an ihr auftreten können.

Wie bereits oben erwähnt gibt es zu jeder Sorte von Ereignissen eine zugehörige Klasse gleichen Namens im Paket `java.awt.event`. Diese ist direkt oder indirekt von `java.awt.AWTEvent` abgeleitet. Abb. 5.4 zeigt die entsprechende Klassenhierarchie. Beim Auftreten eines Ereignisses wird ein Objekt vom Typ der Klasse erzeugt, die der Ereignissorte entspricht, zu der das aufgetretene Ereignis gehört. Dieses Objekt wird beim Aufruf der das Ereignis kommunizierenden Methode als aktueller Parameter übergeben. Dieses Ereignisobjekt enthält zusätzliche Informationen wie z.B. an welcher Komponente das Ereignis aufgetreten ist, d.h. welche Komponente seine Quelle ist (Abfrage mit der Methode `getSource`) oder an welcher Position ein Mausklick erfolgt ist.

**Reagieren auf Ereignisse.** Was passiert, wenn an einer Komponente ein Ereignis auftritt? Wie kann eine GUI auf ein Ereignis reagieren? Das sind die Fragen, die wir im Folgenden klären wollen. Das Lösungskonzept ist einfach:

1. Eine Komponente muss bekannt geben, Ereignisse welcher Ereignissorten an ihr auftreten können.
2. Jedes Objekt, das bestimmte Voraussetzungen erfüllt, kann sich bei Komponenten als *Beobachter* einer oder mehrerer der ihr zugeordneten Ereignissorten registrieren lassen. Im AWT werden diese Beobachter *Listener* genannt<sup>8</sup>.
3. Zum Zweck des Registrierens muss eine Komponente für jede der ihr zugeordneten Ereignissorten (oder für Teilmengen davon<sup>9</sup>) Methoden zur Verfügung stellen, mit Hilfe derer sich an diesen Ereignissen interessierte Beobachter bei ihr registrieren können.

Beobachter  
registrieren

<sup>8</sup>Im Kurs wird der allgemeinere Terminus „Beobachter“ als Übersetzung von *Listener* verwendet, da die *Listener* eine direkte Realisierung des Beobachtermusters darstellen (vgl. Abschn. 3.2.6.4).

<sup>9</sup>Bei einigen Ereignissorten wie z.B. `MouseEvent`s und `WindowEvent`s werden die möglichen Ereignisse in Teilmengen unterteilt, für die je eine eigene Registriermethode zur Verfügung steht. Beobachter, die alle Ereignisse einer in Teilmengen unterteilte Ereignissorte beobachten wollen, müssen sich für jede dieser Teilmengen registrieren.

4. Tritt an einer Komponente  $k$  ein Ereignis der Sorte  $ESEvent$  auf, werden alle Beobachter davon benachrichtigt, die bei  $k$  für  $ESEvent$  registriert sind. Die Benachrichtigung erfolgt durch Aufruf der das Ereignis identifizierenden Methode auf jedem der registrierten Beobachterobjekte. Um einen solchen Aufruf zu ermöglichen, müssen die Beobachter bestimmten Regeln genügen, die weiter unten beschrieben werden.

Abbildung 5.5 illustriert dieses Szenario. Das Button-Objekt  $sf$  repräsentiert die Schaltfläche mit der Beschriftung „zurueck“. Bei  $sf$  sind die Beobachter  $b1$  und  $b2$  registriert. Wird auf die Schaltfläche geklickt, tritt an  $sf$  das Ereignis `actionPerformed` auf. Durch Aufruf der Methode gleichen Namens benachrichtigt  $sf$  daraufhin die beiden Beobachter.

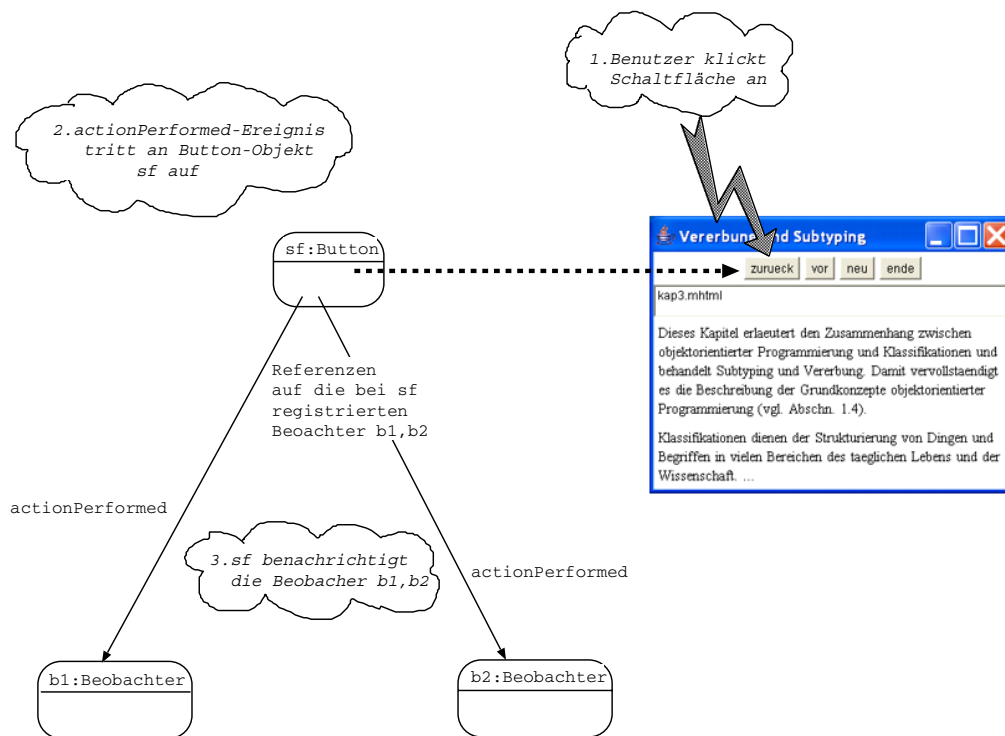


Abbildung 5.5: Eine Schaltfläche benachrichtigt ihre Beobachter

Bevor wir die allgemeine programmtechnische Umsetzung beschreiben, betrachten wir zunächst ein Beispiel. Bei jedem Anklicken einer Schaltfläche soll die Zeichenreihe „Schaltflaeche betaetigt“ ausgegeben werden.

Wie wir bereits erwähnt haben, wird beim Anklicken eines Button-Objekts ein `actionPerformed`-Ereignis generiert, das zur Ereignissorte `ActionEvent` gehört. (Wir werden später darauf eingehen, wie Button-Komponenten bekannt geben, dass an ihnen Ereignisse der Sorte `ActionEvent` auftreten können.) Um das beschriebene Verhalten zu realisieren, muss folgendes gelten bzw. getan werden:



- An der entsprechenden Button-Komponente muss ein Beobachter-Objekt für ActionEvents registriert werden. Objekte, die ActionEvents beobachten sollen, müssen bestimmte Voraussetzungen erfüllen. Sie müssen nämlich die Schnittstelle `ActionListener` implementieren. Dabei besteht diese Schnittstelle nur aus der Methode `void actionPerformed (ActionEvent e);` über die das einzige zur Sorte `ActionEvent` gehörende Ereignis kommuniziert wird.

Beobachter-Objekte für unser Beispiel könnten dann durch folgende Klasse beschrieben werden:

```
class Beobachter implements ActionListener {
    public void actionPerformed((ActionEvent e) {
        System.out.println("Schaltflaeche betaetigt");
    }
}
```

- Damit sich Beobachter von ActionEvents an der Button-Komponente registrieren können, muss die Button-Komponente eine entsprechende Registriermethode zur Verfügung stellen. Diese ist im Fall von ActionEvents:

```
public void addActionListener (ActionListener l).
```

Ein Beobachter *b* vom Typ `Beobachter` kann sich an einem Button-Objekt *sf* also registrieren durch `sf.addActionListener (b);`.

- Beim Auftreten des Ereignisses `actionPerformed` an einer Button-Komponente *sf* werden die an dieser Komponente registrierten Beobachter *b* durch Aufruf von `b.actionPerformed (e)` benachrichtigt. Dabei ist *e* ein `ActionEvent`-Objekt das zusätzliche Informationen zu dem aufgetretenen Ereignis enthält. Für ActionEvents, die an Button-Objekten aufgetreten sind, enthält *e* z.B. Informationen über die Beschriftung des Buttons, einen Zeitstempel, der angibt, wann das Ereignis aufgetreten ist etc. Die Methode `actionPerformed` des Beobachters kann dann alle Aktionen steuern, die der Beobachter bei einem Mausklick auslösen soll. In unserem Fall von Beobachtern des Typs `Beobachter` besteht die Aktion in der Ausgabe des Textes „Schaltflaeche betaetigt“.

Noch offen ist, wie AWT-Komponenten bekannt geben, dass an ihnen ActionEvents auftreten können. Dies teilen sie durch das Vorhandensein der vorgestellten Registriermethode mit.

Das Beispiel demonstriert alle wesentlichen Schritte zur Realisierung einer Ereignissteuerung:

1. Eine Komponente muss bekannt geben, Ereignisse welcher Ereignissorten an ihr auftreten können. Dies tut sie durch Bereitstellen von Registriermethoden, die einem bestimmten Muster folgen. Für eine Ereignissorte *ESEvent* muss die Registriermethode wie folgt aussehen: `addESListener(ESListener l)`.
2. Soll an einer Komponente auf bestimmte Ereignisse der Sorte *ESEvent* reagiert werden, benötigt man ein Beobachter-Objekt, das den Schnittstellentyp mit Namen *ESListener* implementiert<sup>10</sup>. In der Schnittstelle *ESListener* gibt es zu jedem Ereignis *e* der Sorte *ESEvent* genau eine Methode mit Namen *e*. Die Implementierung dieser Methode legt fest, was passieren soll, wenn der Beobachter von dem Ereignis benachrichtigt wird. Typischerweise stößt die Methode *e* eine entsprechende Operation der Anwendung an (vgl. Abschn. 5.3).
3. Mit Hilfe der von der Komponente zur Verfügung gestellten Registriermethode `addESListener` registriert man Beobachter-Objekte von *ESEvents* bei Oberflächenkomponenten.
4. Tritt an einer Komponente *k* ein Ereignis *e* der Sorte *ESEvent* auf, werden die Beobachter-Objekte, die bei *k* für Ereignisse der Sorte *ESEvent* registriert sind, automatisch durch Aufruf der Methode mit Namen *e* benachrichtigt. Dabei wird der Methode als Parameter ein Objekt übergeben, das Informationen zu dem aufgetretenen Ereignis enthält. Die Klassen, die diese Objekte im AWT beschreiben, heißen genau wie die entsprechenden Ereignissorten.

Dieses Konzept zur Ereignissteuerung ist zwar programmtechnisch etwas gewöhnungsbedürftig, andererseits aber sehr flexibel: Beobachter können sich bei mehreren Komponenten gleichzeitig registrieren lassen. Sie können nur einen Teil der Ereignisse beobachten oder sehr viele, indem sie mehrere Listener-Schnittstellen implementieren und sich für jede der zugehörigen (Teil)Ereignissorten registrieren. Eine Komponente kann mehrere Beobachter registrieren. Komponenten können auch selbst als Beobachter fungieren, wenn sie außer den Standardmethoden auch Methoden von Listener-Schnittstellen implementieren. Das Konzept liefert ein typisches Beispiel objektorientierter Programmierung mit Delegation der Aufgaben an bestimmte, weitgehend unabhängige Objekte, die über Nachrichten miteinander kommunizieren.

---

<sup>10</sup>Für *MouseEvents* gibt es zwei Schnittstellen, nämlich *MouseMotionListener* für *mouseDragged*-Ereignisse und *mouseMoved*-Ereignisse sowie *MouseListener* für alle anderen *MouseEvents*. Für *WindowEvents* gibt es drei Schnittstellen: *WindowFocusListener* für *windowGainedFocus*-Ereignisse und *WindowLostFocus*-Ereignisse, *WindowStateListener* für *WindowStateChange*-Ereignisse und *WindowListener* für alle anderen *WindowEvents*.

Die folgenden beiden Beispiele demonstrieren diese Flexibilität. Das erste Beispiel zeigt drei verschiedene Beobachter von ActionEvents, die sich an demselben Button *sf1* registrieren sowie an einem weiteren Button *sf2*. Dabei ist der erste Beobachter vom Typ *Beobachter*, den wir schon kennen gelernt haben.

```
class Beobachter1 implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        String s = e.getActionCommand();
        if (s.equals("OK"))
            System.out.println("OK-Button gedrueckt");
        if (s.equals("Abbruch"))
            System.out.println("Abbruch-Button gedrueckt");
    }
}

class Beobachter2 implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        ...
    }
}

class BeobachterTest {
    public static void main (String[] args) {
        ...
        Button sf1 = new Button("OK");
        Button sf2 = new Button("Abbruch");

        Beobachter b = new Beobachter();
        Beobachter1 b1 = new Beobachter1();
        Beobachter2 b2 = new Beobachter2();

        sf1.addActionListener(b);
        sf1.addActionListener(b1);
        sf1.addActionListener(b2);

        sf2.addActionListener(b);
        sf2.addActionListener(b1);
        sf2.addActionListener(b2);
        ...
    }
}
```

Das nächste Beispiel zeigt eine Komponente, die selbst als Beobachter fungiert. *Frame1* erweitert die Klasse *Frame* aus dem Paket *java.awt*, die Hauptfenster repräsentiert. *Frame1* implementiert zusätzlich die Schnittstellen *ContainerListener* und *WindowListener*. Ein Objekt der Klasse

`Frame1` kann daher `ContainerEvents` und einen Teil der `WindowEvents` behandeln. Ein `Frame1`-Objekt registriert sich selbst als sein eigener Beobachter für an ihm auftretende `Container`- und `WindowEvents`. Soll das Hauptfenster geschlossen werden, reagiert ein `Frame1`-Objekt darauf mit einem Beenden des Programms durch Aufruf von `System.exit(0);`. Wird dem Fenster eine neue Komponente hinzugefügt, wird geprüft, ob es sich dabei um eine `Button`-Komponente handelt. Falls ja, wird ein interner Zähler für alle enthaltenen `Button`-Objekte hochgezählt. Beim Entfernen eines Buttons wird der Zähler um eins erniedrigt. Werden Komponenten zum Fenster hinzugefügt oder aus dem Fenster entfernt, wird durch Aufrufe von `invalidate` und anschließend `validate` erreicht, dass der zum Hauptfenster gehörende `Layout-Manager` die Darstellung neu berechnet und die neue Darstellung angezeigt wird.

```
import java.awt.*;
import java.awt.event.*;

public class Frame1 extends Frame implements
    ContainerListener, WindowListener {

    int countButton = 0;

    Frame1 () {
        this.addContainerListener(this);
        this.addWindowListener(this);
    }

    // Methoden der Schnittstelle ContainerListener
    public void componentAdded (ContainerEvent e) {
        Component o = e.getChild(); // neu eingefuegte Komponente
        if (o instanceof Button) countButton++;
        System.out.println("Button-Zaehler : " + countButton);
        this.invalidate();
        this.validate();
    }

    public void componentRemoved (ContainerEvent e) {
        Component o = e.getChild(); // aktuell entfernte Komponente
        if (o instanceof Button) countButton--;
        System.out.println("Button-Zaehler : " + countButton);
        this.invalidate();
        this.validate();
    }

    // Methoden der Schnittstelle WindowListener
    public void windowClosing (WindowEvent e) {
        System.exit(0);
    }
}
```

```
}  
public void windowClosed (WindowEvent e) {}  
public void windowOpened (WindowEvent e) { }  
public void windowIconified (WindowEvent e) { }  
public void windowDeiconified (WindowEvent e) { }  
public void windowActivated (WindowEvent e) { }  
public void windowDeactivated (WindowEvent e) { }  
}
```

### 5.2.2.5 Programmtechnische Realisierung des AWT im Überblick

Programmtechnisch betrachtet besteht das AWT aus einer Vielzahl von Klassen, die auf mehrere Pakete aufgeteilt sind. Im Mittelpunkt stehen dabei die Pakete `java.awt` und `java.awt.event`. Sie enthalten die Klassen für die Komponenten und die Layout-Manager, die Basisklassen zur Behandlung der Darstellung, insbesondere die Klasse `Graphics`, sowie alle Schnittstellen und Klassen für die Ereignissteuerung. Gemäß unserer Definition stellen die Klassen des AWT ein Programmgerüst dar (vgl. S. 302):

1. Sie bilden ein System, d.h. sie wirken relativ eng zusammen. Diesen Aspekt werden wir im Folgenden kurz vertiefen.
2. Mit Hilfe dieser Klassen lässt sich eine allgemeine, für viele Anwendungsprogramme relevante softwaretechnische Aufgabe lösen, nämlich die Realisierung graphischer Bedienoberflächen.
3. Das System von Klassen ist erweiterbar und anpassbar. Dies werden wir im Rahmen von Abschn. 5.3 anhand einiger Beispiele demonstrieren.

Bevor wir uns im Abschn. 5.2.3 etwas genauer mit einzelnen Klassen des AWT auseinander setzen, wollen wir uns das Zusammenwirken der Klassen nochmals vergegenwärtigen. Komponenten repräsentieren die Teile einer Oberfläche. Sie verweisen auf die `Graphics`-Objekte für die Darstellung. Behälter-Komponenten ermöglichen es, eine GUI aus mehreren Komponenten zusammenzusetzen. Layout-Manager berechnen die Positionen der Komponenten innerhalb eines Behälters. Dazu benötigen sie die Größe der elementaren Komponenten, die wiederum von deren graphischer Darstellung abhängt (z.B. vom verwendeten Font). Die Neuberechnung der graphischen Darstellung der Oberfläche wird großenteils durch Ereignisse gesteuert.

Komponenten besitzen Methoden zum Registrieren von Beobachter-Objekten (z.B. `addActionListener`). Die unterschiedlichen Arten von Beobachter-Objekten werden durch Schnittstellen beschrieben (z.B. `ActionListener`). Diese Schnittstellen legen fest, über welche Ereignisse ein Beobachter benachrichtigt werden will. Der Beobachter muss für jedes Ereignis eine entsprechende Methode zur Verfügung stellen (z.B. `actionPerformed`). Als

Parameter wird dem Beobachter beim Aufruf einer solchen Methode ein Objekt mitgegeben, dessen Typ Subtyp von `AWTEvent` ist. Dieses Objekt enthält Informationen über das aufgetretene Ereignis (z.B. an welcher Komponente das Ereignis aufgetreten ist). Insgesamt entspricht das Zusammenwirken zwischen Komponenten und Beobachtern also genau dem Grundmodell der objektorientierten Programmierung, wobei die Kommunikation durch Methodenaufrufe realisiert ist.

Dieses enge, kooperierende Zusammenwirken der Klassen im AWT spiegelt sich programmtechnisch in der rekursiven, gegenseitigen Abhängigkeit der Typdeklarationen wider: Zum Beispiel besitzt die Klasse `Component` die Methode `addComponentListener` mit einem Parameter vom Typ `ComponentListener`; die Deklaration der Klasse `Component` benutzt also den Schnittstellentyp `ComponentListener`. Letzterer deklariert u.a. die Methode `componentResized` mit einem Parameter vom Typ `ComponentEvent`, hängt also von der Klasse `ComponentEvent` ab. Diese besitzt eine Methode `getComponent`, die ein Ergebnis vom Typ `Component` liefert, so dass sich der Kreis der rekursiven Abhängigkeit schließt.

### 5.2.3 Praktische Einführung in das AWT

Der letzte Abschnitt hat die Konzepte und Struktur des AWT erläutert. Die Kenntnis der konzeptionellen Aspekte reicht aber nicht aus, Oberflächen mit dem AWT zu programmieren. In diesem Abschnitt werden wir wichtige Teile des AWT anhand kleinerer Beispiele genauer behandeln. Damit soll zum einen gezeigt werden, wie die erläuterten Konzepte programmtechnisch umgesetzt wurden, zum anderen wollen wir das Rüstzeug vermitteln, um einfache GUIs entwickeln und sich weitere Aspekte des AWT selbst erschließen zu können. Dazu reicht es in vielen Fällen aus, direkt die Dokumentation des AWT zu benutzen (man braucht sich auch nicht zu scheuen, in die Quellen des AWT hineinzuschauen). Eine gute Beschreibung des AWT findet sich z.B. in [Krü07].

Dieser Abschnitt erläutert den Umgang mit Hauptfenstern, programmtechnische Aspekte der Ereignisbehandlung, einige der elementaren Komponenten, den Mechanismus zum Zeichnen von Komponentendarstellungen und wichtige Eigenschaften der Layout-Berechnung. Schließlich diskutiert er exemplarisch das Erweitern des AWT um eigene Klassen. Bei allen Programmbeispielen gehen wir implizit davon aus, dass die Pakete `java.awt` und `java.awt.event` importiert werden.

#### 5.2.3.1 Initialisieren und Anzeigen von Hauptfenstern

Unser erstes Ziel ist es, die programmtechnischen Aspekte kennen zu lernen, um ein Hauptfenster auf dem Bildschirm erscheinen zu lassen. Hauptfenster

sind Objekte der Klasse `Frame`. Wie bereits angedeutet, spielen sie eine wichtige Rolle beim Zusammenspiel mit dem zugrunde liegenden Fenstersystem. Komponenten, die keine Fenster sind, werden nämlich nur dann auf dem Bildschirm gezeigt, wenn sie in einem Fenster enthalten sind. Das folgende Programm erzeugt ein Hauptfenster auf dem Bildschirm. (Bevor Sie das Programm testen, lesen Sie bitte diesen Unterabschnitt zu Ende!):

```
public class FrameTest {  
  
    public static void main(String args[]) {  
        Frame f = new Frame();  
        f.setSize( 300, 400 );  
        f.setLocation( 100, 100 );  
        f.setVisible( true );  
    }  
}
```

Zunächst wird ein `Frame`-Objekt erzeugt. Beim Erzeugen dieses Objekts wird implizit vom Laufzeitsystem die Ereignisbehandlung gestartet, die quasi parallel zum Benutzerprogramm läuft<sup>11</sup>. Die Ereignisbehandlung bleibt auch dann aktiv, wenn alle Anweisungen des Benutzerprogramms ausgeführt sind. Insbesondere terminiert das obige Programm nicht, nachdem die Methode `main` ausgeführt wurde, sondern muss durch andere Mechanismen beendet werden. Eine Möglichkeit ist es, das Programm von außen abzubrechen, etwa durch `CTRL+ALT+DEL` unter Windows oder durch `CTRL+C` unter Unix (wer mit diesen Mechanismen nicht vertraut ist, sollte obiges Programm nicht testen). Eine elegantere Version behandeln wir im nächsten Unterabschnitt.

Wie alle Komponenten besitzt ein Hauptfenster eine Größe, eine Position und ein Attribut, in dem vermerkt ist, ob die Komponente auf dem Bildschirm angezeigt werden soll oder nicht. Die entsprechenden Methoden zum Setzen dieser Eigenschaften hat `Frame` von der Klasse `Component` geerbt (vgl. Abb. 5.2). Maßangaben sind jeweils in Pixel anzugeben. Beim Setzen der Größe bezieht sich der erste Parameter auf die Breite, der zweite auf die Höhe. Ein Aufruf von `setSize` ist angeraten, da ein `Frame`-Objekt mit der Größe (0,0) initialisiert wird. Die Position einer Komponente lässt sich mit der Methode `setLocation` verändern. Bei einem Hauptfenster bestimmt die Position die Lage des Fensters auf dem Bildschirm; und zwar bezeichnet der erste Parameter die Entfernung der linken oberen Fensterecke vom linken Bildschirmrand, der zweite Parameter die Entfernung vom oberen Rand. Bei elementaren Komponenten bestimmt die Position die Lage der Komponente relativ zur umfassenden Behälter-Komponente. Immer wenn explizit

---


<sup>11</sup>Die explizite Verwendung mehrerer paralleler Ausführungsstränge wird in Kap. 6 behandelt.

mit Maßangaben in Pixeln programmiert wird, gilt es sich bewusst zu machen, dass die Komponenten auf verschiedenen Rechnern je nach Auflösung unterschiedliche absolute Größen (in cm) haben können.

Bei Hauptfenstern wird das Attribut `visible` im Konstruktor mit `false` initialisiert. Um sie sichtbar zu machen, muss dieses Attribut deshalb mit Hilfe der Methode `setVisible` explizit auf `true` gesetzt werden.

### 5.2.3.2 Behandeln von Ereignissen

Um zu studieren, wann an einem Fenster welche `WindowEvents` auftreten, benutzen wir das Programm von Abb. 5.6. Es registriert einen Beobachter für `WindowEvents` bei dem Hauptfenster. Immer wenn der Beobachter benachrichtigt wird, dass am Hauptfenster ein `WindowEvent` aufgetreten ist, gibt er eine entsprechende Meldung aus. Auf diese Weise wird jedes `WindowEvent` am Fenster protokolliert, und man kann z.B. erkennen, wann das Fenstersystem auf dem eigenen Rechner ein Fenster aktiviert bzw. deaktiviert.

Auf die Ereignisse `windowClosing` und `windowClosed` reagiert der Beobachter nicht nur mit Meldungen, sondern er führt auch entsprechende Operationen aus. Im Einzelnen läuft das wie folgt ab: Der Benutzer gibt die Anforderung, Fenster `f` zu schließen (beispielsweise durch Anklicken der rechten oberen Schaltfläche  in einem MS-Windows-Fenster). Daraufhin tritt an `f` ein `windowClosing`-Ereignis auf. Davon wird der registrierte Fenster-Beobachter durch Aufruf der Methode `windowClosing` benachrichtigt. Er gibt die entsprechende Meldung aus und schließt `f` durch Aufruf der Methode `dispose` (`dispose` ist eine Methode der Klasse `Window` die auch dafür sorgt, dass sämtliche von diesem Fenster und der in ihm enthaltenen Komponenten belegten Ressourcen freigegeben werden). Die Referenz auf `f` besorgt er sich dazu vom übergebenen `WindowEvent`-Objekt (vgl. S. 316). Das Schließen von `f` löst das Ereignis `windowClosed` aus. Darauf reagiert der Fenster-Beobachter mit Programmabbruch.

Will man weitere Ereignisse beobachten, muss man entsprechende Beobachter für `ComponentEvents`, `FocusEvents`, `KeyEvents`, `MouseEvents` und `ContainerEvents` implementieren und beim Fenster registrieren. Auf eine detailliertere Darstellung verzichten wir hier. Stattdessen wollen wir uns kurz eine Technik anschauen, mit der Java die Definition von Beobachtern erleichtert. Ein Beobachter, der nur auf ein bestimmtes Ereignis reagieren will, muss trotzdem alle Methoden der entsprechenden Schnittstelle implementieren. Z.B. muss ein Beobachter, der sich nur für `windowClosing`-Ereignisse interessiert, alle sieben Methoden der Schnittstelle `WindowListener` bereitstellen. Da dies oft lästig ist, stellt Java für jede Beobachter-Schnittstelle, die mehr als eine Methode beinhaltet, eine entsprechende sogenannte *Adapterklasse* zur Verfügung, die die Schnittstelle in trivialer Weise implementiert; d.h. in der Adapterklasse hat jede Methode einen leeren Rumpf. Die Klasse



```
class FensterBeobachter implements WindowListener {
    public void windowOpened(WindowEvent e) {
        System.out.println("Fenster geoeffnet");
    }
    public void windowClosing(WindowEvent e) {
        System.out.println("Anforderung Fenster schliessen");
        ((Window)e.getSource()).dispose();
    }
    public void windowClosed(WindowEvent e) {
        System.out.println("Fenster geschlossen");
        System.exit( 0 );
    }
    public void windowIconified(WindowEvent e) {
        System.out.println("Fenster ikonifiziert");
    }
    public void windowDeiconified(WindowEvent e) {
        System.out.println("Fenster deikonifiziert");
    }
    public void windowActivated(WindowEvent e) {
        System.out.println("Fenster aktiviert");
    }
    public void windowDeactivated(WindowEvent e) {
        System.out.println("Fenster deaktiviert");
    }
}

public class FensterBeobachterTest {
    public static void main(String args[]) {
        Frame f = new Frame();
        f.addWindowListener( new FensterBeobachter() );
        f.setSize( 300, 400 );
        f.setLocation( 100, 100 );
        f.setVisible( true );
    }
}
```

Abbildung 5.6: Beobachten von Fensterereignissen

WindowAdapter sieht also wie folgt aus:

```
public abstract class WindowAdapter
    implements WindowListener {
    public void windowOpened(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}
```

Mit Hilfe der Klasse WindowAdapter können wir z.B. sehr knapp eine Beobachterklasse implementieren, deren Objekte nur windowClosing-Ereignisse beachten und auf sie mit Programmabbruch reagieren:

```
class ClosingBeobachter extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit( 0 );
    }
}
```

Diese Klasse benutzen wir, um uns eine Klasse für Hauptfenster zu schreiben, die man schließen kann und in der alle wichtigen Initialisierungen vorgenommen sind. Diese Klasse nennen wir BaseFrame:

```
class BaseFrame extends Frame {
    public BaseFrame() {
        class ClosingBeobachter extends WindowAdapter {
            ...    // wie oben
        }
        addWindowListener( new ClosingBeobachter() );
        setSize( 300, 400 );
        setLocation( 100, 100 );
    }
}

public class BaseFrameTest {
    public static void main(String args[]) {
        Frame f = new BaseFrame();
        f.setVisible( true );
    }
}
```

Da die Klasse `ClosingBeobachter` nur innerhalb von `BaseFrame` verwendet wird, können wir sie auch als anonyme Klasse realisieren. Da diese kompakte Fassung häufig in Java-Programmen verwendet wird, liefern wir auch für sie den Quelltext:

```
class BaseFrame extends Frame {
    BaseFrame() {
        addWindowListener( new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit( 0 );
            }
        } );
        setSize( 300, 400 );
        setLocation( 100, 100 );
    }
}
```

### 5.2.3.3 Elementare Komponenten

Fenster ohne Inhalt sind auf die Dauer wenig spannend. Da ein Fenster auch ein Behälter ist, können wir ihm mit der Methode `add` Komponenten hinzufügen. In diesem Unterabschnitt werfen wir einen Blick auf die elementaren Komponentenklassen `Label`, `Button` und `TextField`. Abbildung 5.7 veranschaulicht die Oberflächen, die wir im Folgenden realisieren.



Abbildung 5.7: Label, Schaltfläche und Textfeld

**Label.** Die Klasse `Label` dient dazu, einen einzeiligen Text anzuzeigen. Hier ist ein Anwendungsbeispiel dazu:

```
public class LabelTest {
    public static void main(String args[]) {
        Frame f = new BaseFrame();
        f.add( new Label("Deutsches Wort fuer Label?") );
        f.setVisible( true );
    }
}
```

Die im Konstruktor übergebene Zeichenreihe wird im Fenster angezeigt (vgl. Abb. 5.7). Der Ort, an dem der Label im Fenster erscheint, wird von dem verwendeten Layout-Manager bestimmt (siehe unten).

**Schaltflächen.** Schaltflächen werden im AWT durch die Klasse `Button` realisiert. Als Anwendungsbeispiel implementieren wir eine Schaltfläche, die in einem Hauptfenster platziert ist und beim Anklicken das Hauptfenster schließt. Damit die Schaltfläche eine typische Darstellung bekommt (vgl. Abb. 5.7), haben wir beim Hauptfenster den Layout-Manager `FlowLayout` eingestellt. Die Ereignisbehandlung entspricht dem Vorgehen, wie wir es im Zusammenhang mit der Klasse `BaseFrame` erläutert haben (vgl. S. 327). Man beachte nochmals, wie der Schaltfläche ein Beobachter-Objekt einer anonymen Klasse hinzugefügt wird:

```
public class ButtonTest {
    public static void main(String args[]) {
        Frame f = new BaseFrame();
        f.setLayout( new FlowLayout() );

        Button b = new Button("Fenster schliessen");
        b.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent e ) {
                System.exit( 0 );
            }
        } );
        f.add( b );
        f.setVisible( true );
    }
}
```

Als weiteres<sup>12</sup> Beispiel für die Erweiterung des AWT realisieren wir eine eigene Klasse für Schaltflächen. Diese Klasse soll `DoButton` heißen und die Syntax für das Hinzufügen der Beobachter erleichtern. Bei ihr sorgt bereits der Konstruktor für die Registrierung eines Beobachters. Der Anwender braucht nur noch die Methode `doAction` zu überschreiben. Hier ist die Klassendeklaration und eine Testanwendung:

```
public class DoButton extends Button {
    public DoButton( String s ) {
        super( s );
        addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent e ) {
                doAction();
            }
        } );
    }
}
```

---

<sup>12</sup>Die Klasse `BaseFrame` war unser erstes Beispiel.

```

        } );
    }
    public void doAction() {}
}

public class DoButtonTest {
    public static void main(String args[]) {
        Frame f = new BaseFrame();
        f.setLayout( new FlowLayout() );

        Button b = new DoButton("Fenster schliessen") {
            public void doAction() {
                System.exit( 0 );
            } };
        f.add( b );
        f.setVisible( true );
    }
}

```

Die Klasse `DoButton` erleichtert in Standardfällen ein wenig das Erzeugen von Schaltflächen. Sie bietet aber vor allem ein gutes Beispiel, um das Zusammenwirken von anonymen Klassen und Überschreiben zu studieren. Darum betrachten wir zwei Varianten dieser Klasse (siehe Abb. 5.8). Die Klasse `DoButton2` besitzt keine parameterlose Methode `doAction`, sondern eine Methode `actionPerformed` mit dem `ActionEvent`-Objekt als Parameter. Die Methode besitzt also die gleiche Signatur wie die Methode in der `ActionListener`-Schnittstelle. Dadurch kann der Anwender von `DoButton2` auf das `ActionEvent`-Objekt zugreifen. Das Beispiel demonstriert auch, wie man innerhalb einer lokalen Klasse (hier eine anonyme Klasse, die die Schnittstelle `ActionListener` implementiert) auf das Objekt der umfassenden Klasse (hier `DoButton2`) zugreifen kann, welches dem `this`-Objekt der inneren Klasse zugeordnet ist (vgl. Abschn. 2.2.2.1, S. 116): Dazu wird dem Bezeichner `this` der Klassenname der umfassenden Klasse vorangestellt.

Die Klasse `DoButton2` legt sofort die äquivalente, aber effizientere Version `DoButton3` nahe, die ein häufig verwendetes Implementierungsmuster benutzt. In ihr wird das Schaltflächenobjekt als sein eigener Beobachter verwendet. Dies ist möglich, da es die Methode `actionPerformed` und damit die `ActionListener`-Schnittstelle implementiert. Diese Version ist effizienter, da sie auf das gesonderte Beobachterobjekt verzichtet und ein Methodenaufruf eingespart wird.

```

public class DoButton2 extends Button {
    public DoButton2( String s ) {
        super( s );
        addActionListener( new ActionListener() {
            public void actionPerformed((ActionEvent e) {
                DoButton2.this.actionPerformed( e );
            }
        } );
    }
    public void actionPerformed( ActionEvent e ) {}
}

public class DoButton3 extends Button
    implements ActionListener {
    public DoButton3( String s ) {
        super( s );
        addActionListener( this );
    }
    public void actionPerformed( ActionEvent e ) {}
}

public class DoButton23Test {
    public static void main(String args[]) {
        final Frame f = new BaseFrame();
        f.setLayout( new FlowLayout() );

        Button b2 = new DoButton2("Fenster schliessen") {
            public void actionPerformed( ActionEvent e ) {
                System.out.println(toString());
                System.exit( 0 );
            }
        };
        f.add( b2 );

        Button b3 = new DoButton3("Schaltflaeche entfernen") {
            public void actionPerformed( ActionEvent e ) {
                System.out.println(toString());
                f.remove( this );
            }
        };
        f.add( b3 );
        f.setVisible( true );
    }
}

```

Abbildung 5.8: Zwei Varianten der Klasse DoButton

**Textfelder.** Ein Textfeld ist eine Komponente, um einen einzeiligen Text zu editieren und eingeben zu können. Im Anwendungsbeispiel haben wir ein Textfeld erzeugt, das mit einer leeren Zeichenreihe initialisiert wird und zwanzig Zeichen breit sein soll. Beim Auftreten eines ActionEvents wird geprüft, ob aktuell im Textfeld die Zeichenreihe "quit" steht. Ist dies der Fall, wird das Programm abgebrochen:

```
public class TextFieldTest {
    public static void main(String args[]) {
        Frame f = new BaseFrame();
        f.setLayout( new FlowLayout() );

        TextField tf = new TextField("",20);
        tf.setEditable( true );
        tf.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent e ) {
                if( ((TextField)e.getSource())
                    .getText().equals("quit") )
                    System.exit( 0 );
            }
        } );
        f.add( tf );
        f.setVisible( true );
    }
}
```

An einem Textfeld kann man verschiedene Ereignisse beobachten. Ein actionPerformed-Ereignis tritt auf, wenn der Benutzer die „Return“- bzw. „Eingabe“-Taste betätigt<sup>13</sup>. Ein textValueChanged-Ereignis tritt auf, wenn der Benutzer den Text im Textfeld verändert. Darüber hinaus kann man natürlich die Ereignisse beobachten, die an allen Komponenten auftreten.

#### 5.2.3.4 Komponentendarstellung selbst bestimmen

Bisweilen möchte man Komponenten verwenden, deren graphische Darstellung man selbst bestimmt bzw. deren Darstellung sich während der Programmlaufzeit verändert (Beispiele: eine Uhr mit Zeigerdarstellung auf der Oberfläche darstellen; ein Balkendiagramm anzeigen). Dazu muss man sich mit zwei Aspekten vertraut machen:

1. Wie kann man überhaupt etwas auf den Bildschirm schreiben bzw. zeichnen; d.h. welche Methoden stellt das AWT dafür zur Verfügung?

<sup>13</sup>Dies ist zumindest das Verhalten der Java-Implementierung, die zum Testen der Programmbeispiele verwendet wurde. Die zugehörige AWT-Dokumentation war diesbzgl. allerdings recht unklar.

## 2. Wie funktioniert der Mechanismus, der im AWT das Zeichnen von Komponenten anstößt und steuert?

Dieser Mechanismus hat mehrere Aufgaben zu bewältigen: Er muss die Komponenten zeichnen, wenn sie das erste Mal auf dem Bildschirm sichtbar werden sollen. Er muss sicherstellen, dass sie neu gezeichnet werden, wenn das Fenster, in dem sich die Komponente befindet, in der Größe verändert wird oder wieder in den Vordergrund kommt, nachdem es verdeckt war. Auch muss gewährleistet werden, dass das Layout eines Behälters, in den eine zusätzliche Komponente eingefügt wurde, neu berechnet und angezeigt wird. Schließlich muss auch die Anwendung selbst in der Lage sein, ein Neuzeichnen anzustoßen.

Auf den Aspekt des Neuzeichnens sind wir bereits im Absatz „Darstellen einzelner Komponenten“ auf S. 312f kurz eingegangen. Immer wenn eine Komponente neu gezeichnet werden muss, wird von der Basisschicht des AWT die Methode `paint` auf der Komponente aufgerufen. Wollen wir etwas dauerhaft in einer Komponente anzeigen, müssen wir `paint` überschreiben. Es ist allerdings nicht ratsam, die Methode `paint` einer beliebigen Komponente zu überschreiben. Beispielsweise stößt `paint` bei Behälter-Komponenten auch das Zeichnen der im Behälter enthaltenen Komponenten an, so dass ein unbedachtes Überschreiben leicht zu unerwünschten Effekten führt. Als Mittel der Wahl zur Realisierung von Komponenten mit selbstgestalteter Darstellung bietet das AWT den elementaren Komponententyp `Canvas` an („canvas“ bedeutet hier soviel wie die Leinwand für ein Ölgemälde).

Die Darstellung von Komponenten auf dem Bildschirm wird mittels eines `Graphics`-Objekts beschrieben (vgl. 5.2.2.3). Das aktuelle `Graphics`-Objekt einer Komponente wird den Methoden `update` und `paint` beim Aufruf als Parameter mitgegeben. Die Klasse `Graphics` besitzt Methoden zum Zeichnen von Bildern, Linien, Ovalen, Polygonen, Polygonzügen, Rechtecken und Zeichenreihen (`drawImage`, `drawLine`, `drawOval`, `drawPolygon`, `drawPolyline`, `drawRect`, `drawRoundRect`, `drawString`). Außerdem gibt es Methoden, um Figuren auszufüllen (z.B. `fillRect`). Ein `Graphics`-Objekt stellt auch den *Graphikkontext* zum Zeichnen zur Verfügung; d.h. die aktuelle Farbe und den aktuellen Font. Mit den Methoden `getColor`, `getFont` bzw. `setColor`, `setFont` können die Farbe und der Font abgefragt und verändert werden. (Darüber hinaus enthält das aktuelle `Graphics`-Objekt die Koordinaten des Bildausschnitts (engl. `clip`), der neu zu zeichnen ist. Die Koordinaten werden z.B. von dem Fenstersystem gesetzt, wenn ein Teil des Fensters in den Vordergrund gekommen ist; sie können aber auch von der Oberfläche selbst gesetzt werden, um einen Zeichenvorgang zu optimieren.)

*Graphik-  
kontext*

Insgesamt erhalten wir folgende prinzipielle Vorgehensweise: Man leitet eine Klasse von `Canvas` ab und überschreibe die Methode `paint`. Als



Beispiel imitieren wir die Funktionalität von `Label` inklusive ihrer Methode `setText`, die hier die Anwendung der Methode `repaint` zum Auslösen eines Neu-Zeichnens demonstriert. (Achtung: Wie in vielen Fällen birgt auch hier das Remake Probleme; siehe unten). Das folgende Programm erzeugt ein Fenster ähnlich dem ersten in Abb. 5.7, wobei die Zeichenreihe "WerKannWas?" angezeigt wird:

```
class LabelCanvas extends Canvas {
    private String word;

    public LabelCanvas( String s ) { word = s; }

    public void paint(Graphics g) {
        g.drawString(word, 3, 200 );
    }
    public void setText (String s) {
        word = s;
        repaint(); // Loest Anzeige des neu gesetzten Textes aus.
    }
}

public class CanvasTest {
    public static void main(String args[]) {
        Frame f = new BaseFrame();
        f.add( new LabelCanvas("WerKannWas?") );
        f.setVisible( true );
    }
}
```

Die Klasse `LabelCanvas` demonstriert zwar, wie man eine Komponente mit eigener Darstellung bauen kann. Sie lässt sich aber nicht in jedem Programmkontext verwenden. Der Grund dafür ergibt sich aus dem Zusammenspiel von elementaren Komponenten und Layout-Managern (wieder ein Beispiel, an dem man erkennen kann, dass das AWT ein Programmgerüst ist, also nicht nur eine Ansammlung kaum zusammenhängender Klassen). Das obige Programm funktioniert, weil der voreingestellte Layout-Manager von `Frame` und damit von `BaseFrame` dem hinzugefügten `LabelCanvas` die gesamte Fläche des Fensters zuteilt. Würde man vorm Hinzufügen der `LabelCanvas`-Komponente den Layout-Manager `FlowLayout` einstellen (siehe S. 329), erhielte die `LabelCanvas`-Komponente die Breite und Höhe null entsprechend der bei `Canvas` voreingestellten Werte und würde somit nicht auf dem Bildschirm erscheinen. Im folgenden Unterabschnitt werden wir uns näher mit diesem Problem und seiner Lösung beschäftigen.

### 5.2.3.5 Layout-Manager: Anordnen von Komponenten

Üblicherweise besteht eine graphische Bedienoberfläche aus mehreren Komponenten. Die Darstellung der gesamten Bedienoberfläche setzt sich dabei aus den Darstellungen der einzelnen Komponenten zusammen. Wie dieses Zusammensetzen erfolgen soll, muss der Oberflächen-Designer beschreiben. In Unterabschn. 5.2.2.3, S. 313, haben wir bereits gesehen, dass das feste Platzieren der einzelnen Komponenten im Allgemeinen wenig sinnvoll ist. Deshalb bietet das AWT sogenannte *Layout-Manager* an, die automatisch die Darstellung eines Behälters  $B$  aus den Darstellungen der in  $B$  enthaltenen Komponenten berechnen. In diesem Unterabschnitt werden wir zwei der fünf Layout-Manager des AWT vorstellen.

**Die Baumstruktur der Komponenten.** Die Komponenten einer GUI sind baumartig strukturiert. Den Blättern des Baumes entsprechen die elementaren Komponenten. Den Knoten des Baumes entsprechen die Behälter-Komponenten. Die Wurzel des Baumes ist ein Fenster. Das Layout der GUI wird rekursiv über diese Baumstruktur berechnet. Jeder Behälter  $B$  besitzt einen Layout-Manager, der die zu  $B$  gehörenden Komponenten in das für  $B$  vorgesehene Rechteck platziert.

Im AWT gibt es drei Arten von Behältern: Fenster, ScrollPanee und Panel (vgl. Abb. 5.2, S. 311). Fenster sind Behälter, die in keinem anderen Behälter enthalten sind (der Versuch, ein Fenster einem Behälter hinzuzufügen, verursacht eine `IllegalArgumentException`). Eine ScrollPane ist ein Behälter für genau eine Komponente. Sie stellt vertikale und horizontale Rollbalken zum Scrollen der Komponentendarstellung zur Verfügung. Zum baumartigen Strukturieren einer GUI dient die Klasse `Panel`.

**Border-Layout.** Beim Border-Layout wird die Fläche des Behälters in fünf Regionen eingeteilt: Center, North, West, South, East. Beim Hinzufügen einer Komponente mittels der Methode `add` legt der Programmierer fest, in welcher Region eine Komponente erscheinen soll. Ein Programmbeispiel macht dies deutlich (bei Fenstern ist Border-Layout voreingestellt):

```
public class BorderLayoutTest {
    public static void main(String args[]) {
        Frame f = new BaseFrame();

        f.add( new Button("Center"), BorderLayout.CENTER );
        f.add( new Button("North"), BorderLayout.NORTH );
        f.add( new Button("South"), BorderLayout.SOUTH );
        f.add( new Button("West"), BorderLayout.WEST );
        f.add( new Button("East"), BorderLayout.EAST );
    }
}
```

```

        f.setVisible( true );
    }
}

```

Werden in eine Region mehrere Komponenten eingefügt, wird die zuletzt eingefügte angezeigt. (Sämtliche im Fenster enthaltenen Komponenten können aber mit Hilfe der Methode `getComponents` ausgelesen werden.)

Bei der Layout-Berechnung muss man zwischen der bevorzugten Größe einer Komponente (engl. preferred size) und der tatsächlichen, vom Layout-Manager bestimmten Größe unterscheiden. Anhand der Layout-Berechnung vom BorderLayout-Manager kann man das recht gut erklären. Dazu zeigt Abbildung 5.9 drei unterschiedliche Layouts für obiges Programm, die durch Verändern der Größe des umfassenden Fensters entstanden sind.

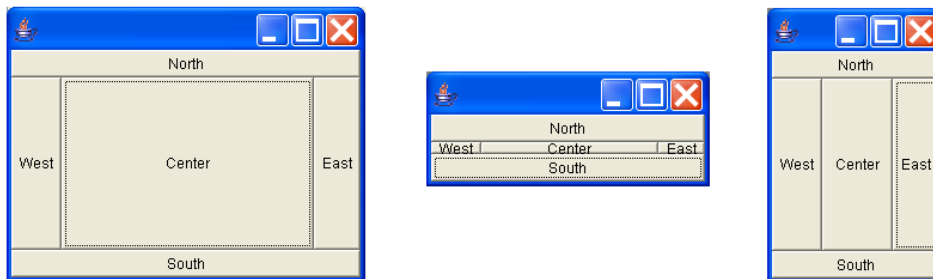


Abbildung 5.9: Border-Layout bei verschiedenen Fenstergrößen

Ausgehend von der Größe des umfassenden Behälters (in der Abbildung ist das der umfassende Frame) bestimmt der BorderLayout-Manager die Größe der fünf enthaltenen Komponenten wie folgt: Die Breite der Nord- und Südkomponenten ist gleich der Behälterbreite, die Höhe der Nord- und Südkomponenten entspricht den von ihnen bevorzugten Höhen. Ebenso entspricht die Breite der West- und Ostkomponenten deren bevorzugten Breiten. Die Höhe der West- und Ostkomponenten und der zentralen Komponente ist die Differenz zwischen der Behälterhöhe und der Summe der Höhen der Nord- und Südkomponenten. Die Breite der zentralen Region ist die Differenz aus der Behälterbreite und der Summe der Breiten der West- und Ostkomponenten. Wie in Abb. 5.9 erkennbar, kann diese Layout-Berechnung dazu führen, dass beispielsweise die Komponente in der zentralen Region nur unvollständig dargestellt wird.

Border-Layout ist gut geeignet, um einfache GUIs zu entwickeln, die aus einem Fenster mit wenigen elementaren Komponenten bestehen. Das ist auch der Grund, warum bei Fenstern Border-Layout voreingestellt ist. Ein zweites, etwas realistischeres Beispiel soll uns dazu dienen, andere Verwendungen des Border-Layouts kennen zu lernen. Dabei werden wir auch weitere Aspekte des AWT veranschaulichen. Abbildung 5.10 zeigt die gewünschte Darstellung der zu realisierenden Oberfläche. Das Programm soll sich wie

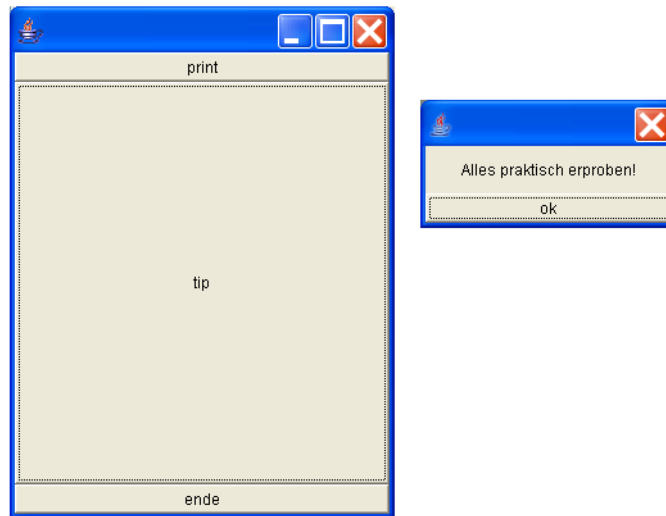


Abbildung 5.10: Oberfläche mit Dialogfenster

folgt verhalten: Zunächst erscheint nur das linke Fenster mit den drei Schaltflächen „print“, „tip“ und „ende“. Anklicken von „print“ soll eine Ausgabe an der Konsole erzeugen. Anklicken von „tip“ soll das Hinweisfenster in der rechten Bildhälfte mit einem kurzen Tipp anzeigen. Während dieses Fenster sichtbar ist, sollen die Schaltflächen des Hauptfensters nicht bedienbar sein. Durch Anklicken von „ok“ soll sich das Hinweisfenster schließen. Mit „ende“ soll das Programm abgebrochen werden können.

Gehen wir zunächst davon aus, dass wir eine Implementierung fürs Hinweisfenster besitzen, die es erlaubt, ein Hinweisfenster zu erzeugen, auf sichtbar bzw. nicht sichtbar zu setzen und den angezeigten Hinweis zu ändern. Mit diesen Voraussetzungen schauen wir uns die Implementierung des Hauptfensters in Abb. 5.11 an: Es werden drei Schaltflächen erzeugt und dem Hauptfenster hinzugefügt; die West- und Ostregion bleibt leer. Interessant ist die Schaltfläche zum Öffnen des Hinweisfensters. Sie enthält die Tipps als String-Feld, einen aktuellen Index in diesem Feld und das Hinweisfenster als Attribut. Beim Erzeugen des Hinweisfensters muss die Referenz auf das Hauptfenster übergeben werden. Da in Klassen, die innerhalb einer Methode `m` deklariert sind, methodenlokale Variable von `m` nur verwendet werden dürfen, wenn sie unveränderlich sind, musste `f` als `final` deklariert werden. Beim Anklicken der tip-Schaltfläche wird der Text im Hinweisfenster eingestellt und angezeigt und der Index weitergezählt.

Die Implementierung von Hinweisfenstern bietet ein weiteres Beispiel für die Benutzung des Border-Layouts und illustriert die Benutzung der AWT-Klasse `Dialog`. Mit Dialogfenstern kann man die Eingabe an anderen Fenstern blockieren; d.h. der Benutzer muss sich zunächst mit dem Dialogfenster beschäftigen, bevor er wieder Zugang zu den anderen Bedien- und Eingabe-

```

public class HinweisFensterTest {
    public static void main(String args[]) {
        final Frame f = new BaseFrame();

        Button printBt = new DoButton("print") {
            public void doAction() {
                System.out.println("Fernleere?");
            }
        };
        Button tipBt = new DoButton("tip") {
            String [] tips = {
                "Alles praktisch erproben!",
                "Nicht verzagen!",
                "Nachdenken macht schoen!"    };
            int aktIndex = 0;
            HinweisFenster hf = new HinweisFenster(f,"init");

            public void doAction() {
                hf.setText( tips[aktIndex] );
                hf.setVisible( true );
                aktIndex = (aktIndex+1) % 3;
            }
        };
        Button endeBt = new DoButton("ende") {
            public void doAction() {
                System.exit( 0 );
            }
        };

        f.add( printBt, BorderLayout.NORTH );
        f.add( tipBt,    BorderLayout.CENTER );
        f.add( endeBt,   BorderLayout.SOUTH );
        f.setVisible( true );
    }
}

```

Abbildung 5.11: Programm zum Hauptfenster von Abb. 5.10

elementen seiner Anwendung bekommt. Man spricht in diesem Zusammenhang auch von einem *modalen Dialog*. Typischerweise werden Dialogfenster benutzt, um Hinweise anzuzeigen, die der Benutzer lesen soll, bevor er weiter arbeitet, oder um Eingaben anzufordern, die für die Fortsetzung erforderlich sind. Bei Erzeugung wird dem Dialogfenster eine Referenz auf das Fenster übergeben, das es ggf. blockiert, sowie ein boolescher Wert, der bestimmt, ob der Dialog modal sein soll oder nicht.

*modaler  
Dialog*

```
public class HinweisFenster extends Dialog {
    Label hinweisLabel;

    public HinweisFenster( Frame f, String hinw ) {
        super( f, true );    // true bedeutet modaler Dialog
        hinweisLabel = new Label(hinw, Label.CENTER);
        add( hinweisLabel, BorderLayout.CENTER );

        Button ok = new DoButton("ok") {
            public void doAction() {
                HinweisFenster.this.setVisible(false);
            }
        };
        add( ok, BorderLayout.SOUTH );

        setSize( 200, 100 );
        setLocation( 400, 200 );
    }
    public void setText( String s ) {
        hinweisLabel.setText( s );
    }
}
```

Abbildung 5.12: Die Klasse HinweisFenster

Abbildung 5.12 zeigt eine mögliche Implementierung der Klasse HinweisFenster. Zum vollständigen Verständnis sind folgende Bemerkungen hilfreich: Bei dem Erzeugen eines Label-Objekts kann man angeben, ob der Text in der Komponente zentriert, links- oder rechtsbündig angezeigt werden soll (durch Angabe der Konstanten `Label.CENTER`, `Label.LEFT` bzw. `Label.RIGHT`). Um in der `doAction`-Methode auf das Hinweisfenster zugreifen zu können, muss man den Bezeichner `this` entsprechend qualifizieren (vgl. die Diskussion im Zusammenhang mit der Klasse `DoButton` auf S. 329f); andernfalls würde er sich auf die `DoButton`-Komponente beziehen. Die Methode `setText` stützt sich auf die entsprechende, vordefinierte Methode der Komponentenkasse `Label`, die dem Label einen neuen Text zuordnet.

**Flow-Layout.** Besitzt ein Behälter einen FlowLayout-Manager, werden die eingefügten Komponenten der Reihe nach von links nach rechts angeordnet. Passt eine Komponente nicht mehr in die aktuelle Zeile, wird in der nächsten Zeile fortgefahren. Wir demonstrieren den FlowLayout-Manager anhand eines Programms, das einen Text aus einer Datei liest, diesen in Worte zerteilt, für jedes Wort ein Label erzeugt und diese Label der Reihe nach einem Panel hinzufügt. Beim Verändern der Fenstergröße sorgt der FlowLayout-Manager dann für den Zeilenumbruch des Textes (siehe Abb. 5.13). Das Programm dazu bietet Abbildung 5.14.

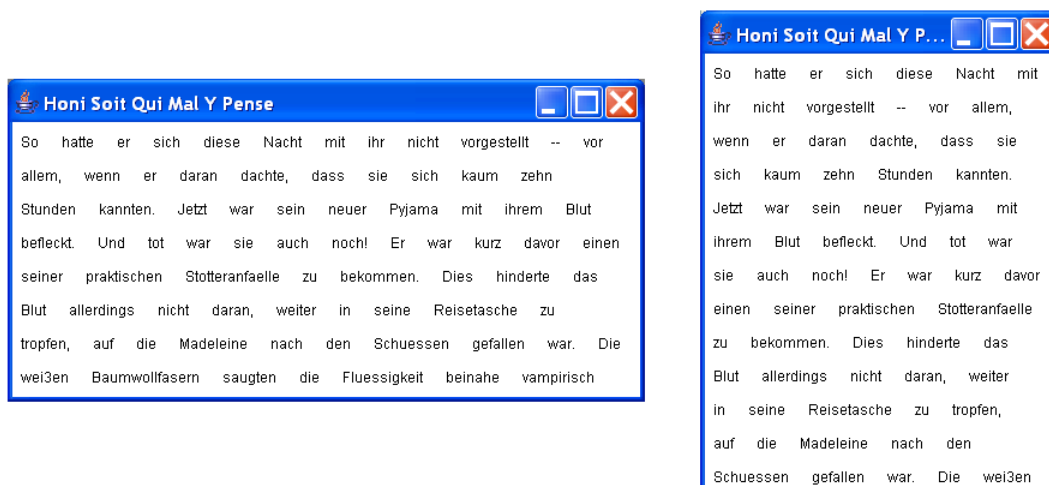


Abbildung 5.13: Flow-Layout bei verschiedenen Fenstergrößen

```
import java.util.StringTokenizer;

public class FlowLayoutTest {
    public static void main(String args[]) throws Exception {
        Frame f = new BaseFrame();
        f.setTitle("Honi Soit Qui Mal Y Pense");

        String text = DateiZugriff.read("Textdatei.txt");
        StringTokenizer st = new StringTokenizer( text, " \t\n\r");
        Panel p = new Panel( new FlowLayout(FlowLayout.LEFT) );

        while( st.hasMoreTokens() ){
            String token = st.nextToken();
            p.add( new Label( token ) );
        }
        f.add( p );
        f.setVisible( true );
    }
}
```

Abbildung 5.14: Programm zur Illustration von Flow-Layout

Das Programm benutzt die Klasse `DateiZugriff` aus Kap. 4 zum Einlesen des Textes und die Klasse `StringTokenizer` aus dem Paket `java.util` zum Zerteilen des Textes in Token bzw. Worte. Als Trenner zwischen Worten werden das Leerzeichen, das Tabulatorzeichen sowie die Zeichen zum Zeilenumbruch verwendet. Das Panel wird mit einem `FlowLayout`-Manager versorgt, der die Zeilen linksbündig anordnet. Im folgenden Unterabschnitt „Erweitern des AWT“ werden wir uns die Berechnung des `FlowLayout`s etwas genauer anschauen.

**Die anderen Layout-Manager.** Außer dem `Border`- und `Flow`-Layout unterstützt das AWT drei weitere Layout-Arten: `Card`-Layout gestattet es, mehrere Komponenten übereinander zu legen. Am oberen Rand des Behälters befindet sich für jede Komponente eine Schaltfläche, mit der sich der Benutzer jeweils die entsprechende Komponente in den Vordergrund holen kann. `Grid`-Layout dient dazu, Komponenten in einer (zweidimensionalen) Tabelle anzuordnen. Für komplexere Layouts steht der `GridBagLayout`-Manager zur Verfügung.

#### 5.2.3.6 Erweitern des AWT

In diesem Unterabschnitt wollen wir eine komplexere Erweiterung des AWT demonstrieren. Dabei werden wir insbesondere das Zusammenspiel zwischen der bevorzugten Größe einer Komponente und der vom Layout-Manager bestimmten aktuellen Größe kennen lernen – ein weiteres Beispiel für die enge Verflechtung der Klassen des AWT.

**Festlegen der bevorzugten Größe.** Das Layout des Textes in Abb. 5.13 ist wenig zufriedenstellend. Insbesondere sind die Abstände zwischen den Worten zu groß. Die erste Idee, um hier Abhilfe zu schaffen, könnte sein, statt im Programm von Abb. 5.14 in der Schleife ein `Label` zur Darstellung eines Wortes zu benutzen, eine Komponente vom Typ `LabelCanvas` (vgl. S. 334) einzusetzen. Das Ergebnis wäre ein leeres Fenster. Der Grund dafür ist, dass der `FlowLayout`-Manager bei der Layout-Berechnung die Komponenten mittels der Methode `getPreferredSize` nach ihrer bevorzugten Größe befragt und diese Methode in der Klasse `LabelCanvas` nicht geeignet angepasst wurde (die von `Canvas` geerbte Methode liefert als Größe `(0, 0)` zurück).

Im Allgemeinen reicht es nicht aus, nur die bevorzugte Größe anzupassen, da Layout-Manager auch die minimale Größe einer Komponente mittels der Methode `getMinimumSize` benutzen. Deshalb empfiehlt es sich immer, zumindest für eine richtige Einstellung der bevorzugten und der minimalen Größe zu sorgen. Abbildung 5.15 zeigt hierfür eine mögliche Lösung. Dabei ist die Klasse `WordComponent` der Klasse `Label` vergleichbar, erzeugt allerdings um die dargestellten Worte herum keinen so großen Rand.



```

public class WordComponent extends Canvas {
    private String word;
    private int hoffset;
    private Dimension size;

    public WordComponent( String s ) {
        word = s;
        setFont( new Font("Serif",Font.PLAIN,14) );
        FontMetrics fm = getFontMetrics( getFont() );
        int descent = fm.getMaxDescent();
        int height = fm.getHeight();
        size = new Dimension( fm.stringWidth(s), height );
        hoffset = height-descent;
    }
    public Dimension getMinimumSize()    { return size; }
    public Dimension getPreferredSize() { return size; }

    public void paint(Graphics g){
        g.drawString(word,0,hoffset);
    }
}

```

Abbildung 5.15: Komponente zum Anzeigen eines Wortes

Die Größe von Komponenten wird mittels Objekten des Typs `Dimension` verwaltet und beinhaltet Höhe und Breite. Die Größe einer Wortkomponente wird über die Eigenschaften des eingestellten Fonts berechnet und im Attribut `size` gespeichert. Im Attribut `hoffset` wird der Abstand der Grundlinie, auf der geschrieben werden soll, vom oberen Rand der Komponente festgehalten. Dieser Abstand ergibt sich aus der Differenz zwischen dem Zeilenabstand und der maximalen Unterlänge des Fonts, d.h. der Größe von Buchstabenteilen unterhalb der Grundlinie (`descent`). Die Verwendung von `WordComponent` statt `Label` in der Klasse `FlowLayoutTest` von Abb. 5.14 führt zu dem verbesserten Layout, wie es in Abb. 5.16 zu sehen ist.

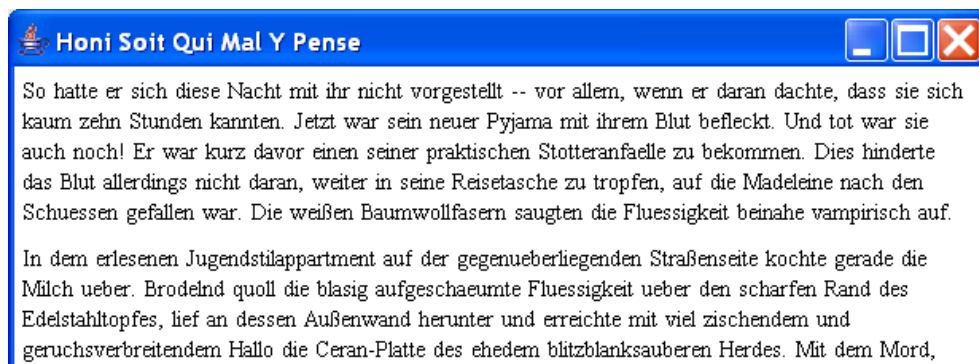


Abbildung 5.16: Vom TextLayout-Manager berechnetes Layout

**Entwickeln von Layout-Managern.** An einem einfachen Beispiel wollen wir abschließend wichtige programmtechnische Aspekte der Layout-Berechnung studieren. In Abschn. 2.1.3, S. 86, hatten wir die Zeichenreihe "<BR>" verwendet, um den Zeilenumbruch in einem Text zu markieren. Hier wollen wir einen TextLayout-Manager entwickeln, der in der Lage ist, Zeilenumbruchmarkierungen zu beachten. Da die Zeilenumbruchmarkierungen Teil des Behälters sein müssen, für den das Layout zu berechnen ist, führen wir die trivialen, nicht sichtbaren Komponenten `BrMark` ein:

```
public class BrMark extends Component {  
    public BrMark(){ setVisible( false ); }  
}
```

`BrMark`-Komponenten werden an den Stellen eines Behälters mit `FlowLayout` eingefügt, an denen ein Zeilenumbruch erfolgen soll. Da der `FlowLayout`-Manager diesen Zeilenumbruch nicht ausführen würde, müssen wir ihn erweitern. Dazu überschreiben wir die Methode `layoutContainer`, die die Positionierung der Komponenten eines Behälters durchführt. Abbildung 5.17 zeigt eine mögliche Lösung, die wir im Folgenden erläutern werden.

Zunächst werden zwei Größen festgelegt: `gap` legt insbesondere den Zwischenraum zwischen dem Behälterrand und dem ersten Wort bzw. zwischen zwei Worten fest; `minrow` bezeichnet die minimale Zeilenhöhe. Die aktuelle Zeilenhöhe wird mit der minimalen initialisiert. Dann wird die Breite des Behälters ermittelt, für den das Layout zu berechnen ist. Dabei wird die Breite eines möglichen Behälterrahmens abgezogen (die Methode `getInsets` liefert die dafür notwendigen Informationen). Nach diesen Vorbereitungen wird die aktuelle Position im Behälter initialisiert: die x-Koordinate an den linken Rand; die y-Koordinate ein Stück unterhalb des oberen Rands. Die y-Koordinate innerhalb von Behältern wächst nämlich immer von oben nach unten.

Der Reihe nach wird dann für die Komponenten `m` des Behälters folgendes durchgeführt: Ist `m` eine Zeilenumbruchmarkierung, wird der aktuelle x-Wert auf den Anfang der Zeile gesetzt, der aktuelle y-Wert um die ermittelte Zeilenhöhe inkrementiert und die Zeilenhöhe neu initialisiert. Andernfalls setzt der Layout-Manager die aktuelle Größe der Komponente auf deren bevorzugte Größe. Wenn in der Zeile bereits Komponenten platziert sind (`x!=0`) und die nächste Komponente nicht mehr in die Zeile passt (`(x+width+gap)>maxwidth`), wird ein Zeilenumbruch vorgenommen. Im Übrigen wird ein horizontaler Zwischenraum eingefügt (`x+=gap`), `m` an die aktuelle Position platziert, die x-Position um die Breite von `m` verschoben und die Zeilenhöhe auf das Maximum der bisherigen Höhe und der Höhe von `m` festgelegt.

```

public class TextLayout extends FlowLayout {

    public TextLayout() {
        super( FlowLayout.LEFT );
    }

    public void layoutContainer(Container target) {
        final int gap = 5;           // Zwischenraum
        final int minrowh = 10;      // minimale Hoehe einer Zeile
        int rowh = minrowh;          // Zeilenhoehe initialisieren

        Insets insets = target.getInsets();
        int maxwidth = target.getWidth()
            - (insets.left + insets.right + 2*gap);
        int nmembers = target.getComponentCount();
        int x = 0, y = insets.top + gap;

        for (int i = 0 ; i < nmembers ; i++) {
            Component m = target.getComponent(i);
            if( m instanceof BrMark ) {
                x = 0;
                y += rowh;
                rowh = minrowh;
            } else {
                Dimension d = m.getPreferredSize();
                int width  = d.width;
                int height = d.height;
                m.setSize( width, height );

                if((x != 0) && ((x+width+gap) > maxwidth)) {
                    x = 0;
                    y += rowh;
                    rowh = minrowh;
                }
                x += gap;
                m.setLocation(x, y);
                x += width;
                rowh = Math.max(rowh, height);
            }
        }
    }
}

```

Abbildung 5.17: Layout-Berechnung mit Zeilenumbruch

```
public class TextLayoutTest {
    public static void main(String args[]) throws Exception {
        Frame f = new BaseFrame();
        f.setTitle("Honi Soit Qui Mal Y Pense");

        String text = DateiZugriff.lesen("Textdatei.txt");
        StringTokenizer st = new StringTokenizer( text, " \n\r\t");
        Panel p = new Panel( new TextLayout() );

        while( st.hasMoreTokens() ){
            String token = st.nextToken();
            if( token.equals("<BR>") ) {
                p.add( new BrMark() );
            } else {
                p.add( new WordComponent( token ) );
            }
        }
        f.add( p );
        f.setVisible( true );
    }
}
```

Abbildung 5.18: Anwendung des TextLayouts

Eine Anwendung des TextLayouts zeigt Abbildung 5.18. Immer wenn in der Textdatei das Token "<BR>" gefunden wird, wird in der Behälterkomponente eine Zeilenumbruchsmarkierung eingefügt. Da auch leere Zeilen zu einem Umbruch führen, kann man einen Absatz durch Eingabe von "<BR> <BR>" erzeugen. Abbildung 5.16 zeigt das entstehende Layout eines Texts mit Absatz. Weitere Beispiele haben wir bereits in Abschn. 2.1.3, Abb. 2.5, gesehen.

### 5.2.3.7 Rückblick auf die Einführung ins AWT

In diesem Abschnitt haben wir einige praktische Aspekte des AWT behandelt, insbesondere um eine programmtechnische Grundlage für die Realisierung von GUIs zu schaffen. Jede Komponente ist direkt oder indirekt in einem Fenster enthalten. Ein Fenster ist entweder ein Hauptfenster oder ist direkt oder indirekt einem Hauptfenster zugeordnet. Ereignisse treten an Komponenten auf und werden an alle Beobachter gemeldet, die bei der betroffenen Komponente für die entsprechende Ereignissorte registriert sind. Einige der elementaren Komponententypen wurden vorgestellt, und es wurde gezeigt, wie man die Darstellung einer Komponente selbst bestimmen kann. Schließlich wurden Mechanismen erläutert, mit denen die Anordnung der Komponenten in Behältern bestimmt wird.

Die Einführung in das AWT illustriert zudem an einem konkreten Bei-

spiel, was man unter einem Programmgerüst versteht. Insbesondere sollte deutlich geworden sein, wie man mit Hilfe vieler zusammenhängender Klassen die Kernfunktionalität einer komplexeren softwaretechnischen Aufgabe realisieren kann.

## 5.3 Anwendung von Programmgerüsten

Dieser Abschnitt geht auf die systematische Anwendung von Programmgerüsten ein. Er erläutert zunächst den Zusammenhang zwischen Programmgerüsten und Software-Architekturen<sup>14</sup> bzw. Architekturmustern.

Anschließend bespricht er das Zusammenspiel von Programmgerüst, Architektur und Entwicklungsmethodik anhand eines ausführlicheren Beispiels, nämlich der Entwicklung eines Browsers mit graphischer Bedienoberfläche (Abschn. 5.3.2).

Im Kontext dieses Kurses wollen wir unter einer Software-Architektur den Aufbau eines Systems aus Komponenten und Teilsystemen verstehen sowie deren Beziehungen zueinander. Beziehungen können u.a. die Zuordnung zu bestimmten Aufgaben betreffen und/oder das Kommunikationsverhalten. Z.B. können Komponenten in Schichten organisiert sein, wobei jede Schicht nur mit der direkt über ihr und unter ihr liegenden Schicht kommunizieren kann. Eine solche Anordnung ist z.B. als Schichtenarchitektur bekannt und stellt ein Architekturmuster dar. Eine andere geläufige Architektur ist die Client-Server Architektur. Im Zusammenhang mit dem AWT interessiert uns insbesondere die folgende, häufig verwendete Architektur zur Realisierung von graphischen Bedienoberflächen: die Model-View-Controller (MVC) Architektur. Diese Architektur werden wir in Abschnitt 5.3.1 noch näher kennen lernen.

### 5.3.1 Programmgerüste und Software-Architekturen

Gerüste bieten die programmtechnische Basis zur Bewältigung bestimmter softwaretechnischer Aufgabenstellungen. Um flexibel auf unterschiedliche Anforderungen reagieren zu können, soll von den Gerüsten dabei nicht bis ins Detail vorgegeben werden, wie die Software-Architektur zur Lösung der Aufgaben auszusehen hat und mit welchen Methoden die Lösung zu realisieren ist. Beispielsweise bietet auch das AWT etliche Freiheitsgrade beim Entwurf und der Implementierung graphischer Bedienoberflächen. (Für eine genauere Diskussion dieses Aspekts stehen uns hier keine hinreichend um-

<sup>14</sup>Der Begriff der Software-Architektur ist noch immer nicht klar umrissen. Verschiedene Autoren definieren diesen Begriff recht unterschiedlich. An über den hier vorgestellten Abriss hinausgehenden Konzepten Interessierte seien auf den Kurs „Software-Architektur“ verwiesen.

fangreichen Beispiele zur Verfügung; eine kleine Idee kann vielleicht die Behandlung der Versionen von `DoButton` liefern; vgl. S. 330f.)

Andererseits ist es bei der Bewältigung softwaretechnischer Aufgabenstellungen oft sehr hilfreich, sich an bewährten Architekturen bzw. Architekturmustern zu orientieren. Von guten Programmgerüsten wird man also erwarten, dass sie die Realisierung bewährter Architekturen besonders gut unterstützen. Dieses Zusammenspiel zwischen Programmgerüsten und Software-Architekturen wollen wir in diesem Abschnitt am Beispiel der MVC-Architektur und dem AWT studieren. Dazu erläutern wir zunächst in Kürze, was unter der MVC-Architektur zu verstehen ist, und diskutieren dann das Verhältnis zum AWT.

**Die Model-View-Controller-Architektur.** Die Realisierung leistungsfähiger Bedienoberflächen wird schnell zu einer komplexen softwaretechnischen Aufgabe. Sie beinhaltet die Gestaltung der graphischen Oberflächen und die Steuerung des Dialogs zwischen Benutzer und Anwendung. Sie sollte dementsprechend mit konzeptionell nebenläufigen Ereignissen umgehen können und muss die Konsistenz zwischen Anwendung und ggf. mehreren Darstellungen gewährleisten (vgl. Abschn. 5.2.1). Um diese softwaretechnischen Schwierigkeiten beherrschbar zu machen, ist es in vielen Fällen hilfreich, eine graphische Bedienoberfläche in mehrere Teile mit klar definierten Schnittstellen zu zerlegen. Der klassische Ansatz dafür, die sogenannte *MVC-Architektur*, die die architektonische Grundlage des Programmgerüsts zur GUI-Entwicklung in Smalltalk bildet, sieht folgende Zerlegung vor:

MVC-  
Architektur

- **Model/Anwendungsschnittstelle:** Das „Model“ repräsentiert die gesteuerte Anwendung aus Sicht der GUI, bildet also die Schnittstelle zur Anwendung. Der Einfachheit halber werden wir häufig nicht explizit zwischen Anwendung und Anwendungsschnittstelle unterscheiden. Die Anwendungsschnittstelle besteht aus einer Reihe von Operationen, mittels derer die Anwendung von der Bedienoberfläche gesteuert werden kann. Sie meldet Zustandsänderungen an die anderen Oberflächenteile und ermöglicht diesen, bestimmte Zustandsdaten auszulesen.
- **View/Darstellung:** Als „View“ werden die Programmteile bezeichnet, die die graphische Darstellung der Eingabe- und Steuerungskomponenten sowie des Zustands der Anwendung auf dem Bildschirm besorgen. Die Darstellung muss bei Änderungen in der Anwendung entsprechend aktualisiert werden.
- **Controller/Steuerung:** „Controller“ steuern die Interaktion zwischen der Darstellung und der Anwendung. Sie nehmen Benutzereingaben, wie Mausklicks und Tastatureingaben, entgegen, stellen die Verbindung zwischen Bedienelementen der Oberfläche und Operationen der Anwendung her, verwalten Eingabeparameter und lösen die Operationen

in der Anwendung aus. Benutzereingaben können sich auch nur auf die Darstellung beziehen, etwa Verschieben von Oberflächenelementen oder Ändern von graphischen Aspekten (Farbe, verwendeter Font). Derartige Eingaben veranlassen üblicherweise eine Aktualisierung der Darstellung, ohne dass die Anwendung davon betroffen ist.

Das Zusammenwirken dieser Teile in der MVC-Architektur wird in Abb. 5.19 illustriert.

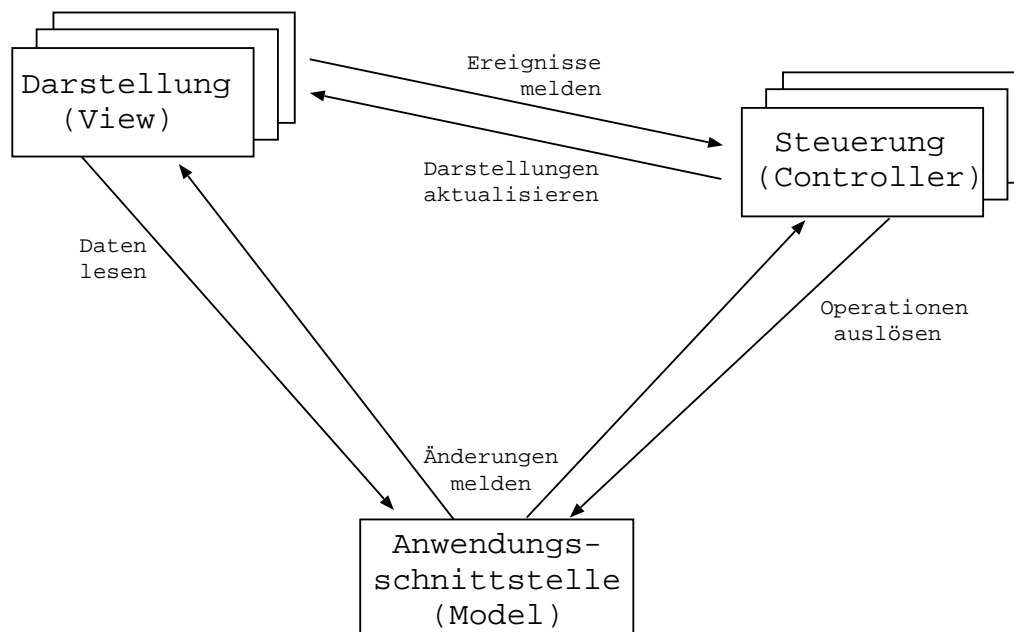


Abbildung 5.19: MVC-Architektur

Wie in der Abbildung angedeutet, können zu einer Anwendung mehrere Darstellungen gehören. Zum Beispiel unterstützen es viele Editoren, mehrere Fenster auf einen Text zu öffnen. Bei Änderungen am Text muss dann die Darstellung in allen Fenstern aktualisiert werden. Ein anderes Beispiel ist die unterschiedliche Darstellung von Daten, etwa in Tabellenform und in Form einer Kurve.

Üblicherweise hat jede Darstellung ihre eigene Steuerungskomponente. Andererseits kann es auch sinnvoll sein, den verschiedenen Komponenten einer Darstellung jeweils eigene Steuerungen zuzuordnen oder eine Steuerung übergreifend für mehrere Darstellungen einzusetzen.

Die MVC-Architektur tritt in vielen Varianten auf. (Zum Beispiel wird häufig keine strikte Trennung zwischen Darstellung und Steuerung realisiert.) Diese Variationsbreite ist eine Stärke der MVC-Architektur. Sie bietet damit nämlich eine flexible Richtschnur für den Entwurf interaktiver Programme mit unterschiedlichen Anforderungen. Interaktive Programme wer-

den in weitgehend unabhängige Teile zerlegt, die miteinander über eine kleine Schnittstelle mittels Nachrichten kommunizieren. Insbesondere kann eine Anwendung mehrere Darstellungen besitzen. Weitere Darstellungen können ohne Änderung der Anwendung hinzugefügt werden.

**Zusammenhang zwischen MVC und AWT.** Die MVC-Architektur bildet den Hintergrund vieler Programmgerüste zur Oberflächenprogrammierung und hat auch den Entwurf des AWT erheblich beeinflusst. Das hat dazu geführt, dass das AWT eine gute Basis bietet, um Bedienoberflächen gemäß der MVC-Architektur zu realisieren. Insbesondere kann mit dem Beobachter-Konzept eine programmtechnisch saubere Trennung zwischen Darstellung und Steuerung umgesetzt werden. Die MVC-Architektur kann also als Entwurfsrichtlinie für die Oberflächenentwicklung mit dem AWT fungieren. Andererseits bietet das AWT ausreichend Flexibilität, um auch andere Architekturen bzw. Architekturmuster zu realisieren. Als kleines Beispiel sei wieder auf die Version der Klasse `DoButton` verwiesen (vgl. S. 330f), bei der Darstellung und Steuerung durch ein Objekt implementiert werden.

### 5.3.2 Entwicklung graphischer Bedienoberflächen

Die Entwicklung von GUIs ist im Allgemeinen eine anspruchsvolle Aufgabe und zwar sowohl aus ergonomischer Sicht (wie organisiert man eine GUI so, dass sie möglichst einfach und komfortabel zu bedienen ist) als auch aus softwaretechnischer Sicht. Wir konzentrieren uns hier auf die softwaretechnischen Aspekte. Die softwaretechnische Komplexität hat im Wesentlichen zwei Ursachen:

1. Eine GUI ermöglicht dem Benutzer zu einem Zeitpunkt viele unterschiedliche Operationen durchzuführen. Daraus resultiert eine große Anzahl erlaubter Dialogabläufe.
2. Der graphischen Oberfläche liegt häufig ein komplexes Objektgeflecht zugrunde, das bei Zustandsänderungen in korrekter Weise zu aktualisieren ist.

Dieser Abschnitt führt in eine Entwicklungsmethodik für GUIs ein und demonstriert sie an einem kleinen, aber nicht-trivialen Beispiel. Als Grundlage orientieren wir uns dabei an der MVC-Architektur. Die verwendete *Entwicklungsmethodik* geht von der Anwendung aus und besteht aus drei Schritten.

- Im ersten Schritt wird die Schnittstelle der Anwendung und die Dialogführung festgelegt.
- Im zweiten Schritt wird die Darstellung entwickelt.



- Im dritten Schritt werden die Anwendung und die Darstellung durch eine Steuerung verbunden.

Als Beispiel wollen wir einfache Browser mit einer graphischen Bedienoberfläche entwickeln. Die Funktionalität der Browser soll ähnlich wie in Kap. 2 und 3 sein. Allerdings soll die dort verwendete Steuerung über die Konsole durch eine graphische Bedienoberfläche ersetzt werden.

Die zentrale Steuerung der Browser über die Konsole war aus objektorientierter Sicht ein Fremdkörper. Das objektorientierte Grundmodell legt es nahe, die Eingaben und Nachrichten direkt an die Browser-Objekte zu schicken. Da für die Ein- und Ausgabe aber nur die Konsole zur Verfügung stand, waren wir auf die eher prozedural ausgelegte Steuerung angewiesen (vgl. Abb. 3.13, S. 225). Mit Hilfe des Kommandos 'w' musste man zunächst schrittweise zum aktuellen Browser wechseln und konnte dann die entsprechenden Eingaben an diesen Browser richten. Eine derartige zentrale Steuerung ist nicht nur bedienungsunfreundlich, sondern bringt auch softwaretechnische Nachteile mit sich. So musste die Steuerung bei der Spezialisierung der Browserklasse vollständig neu geschrieben werden (vgl. Abschn. 3.12, S. 223f). Demgegenüber erlaubt eine graphische Oberfläche die Realisierung eines klassischen objektorientierten Entwurfs: Der Benutzer kann die Browser direkt über Nachrichten und Eingaben ansprechen und steuern.

Im Folgenden beschreiben wir zunächst die Anforderungen an die Browser und ihre Oberfläche und gehen dann auf die drei Entwicklungsschritte ein.

#### 5.3.2.1 Anforderungen

Die Browser sollen Folgendes leisten: Sie sollen W3-Seiten, die textuell in Dateien abgelegt sind, laden und anzeigen können. Dazu soll der Benutzer den Dateinamen in einem Fenster eingeben können. Falls der Zugriff auf die Datei nicht erfolgreich war, soll der Benutzer darüber mittels eines Dialogfensters informiert werden. Die textuelle Repräsentation der W3-Seiten basiert auf einem vereinfachten, ganz kleinen Ausschnitt von HTML (vgl. Abschn. 2.1.3, S. 86), den wir MHTML<sup>15</sup> nennen wollen. MHTML kennt drei Sprachelemente: den Titel einer W3-Seite, die Zeilenumbruchmarkierung und sogenannte *Hypertext-Referenzen*, mit denen man sich auf andere W3-Seiten beziehen kann. Als Beispiel betrachten wir die textuelle Repräsentation der W3-Seite kap2 von Abb. 2.5, S. 87 (aus Kompatibilitätsgründen mit HTML verzichten wir dabei auf die Behandlung von Umlauten):

*Hypertext-  
Referenz*

---

<sup>15</sup>MHTML-Seiten sind mit jedem HTML-fähigen Browser lesbar, also etwa mit dem Navigator von Netscape oder dem InternetExplorer von Microsoft. Allerdings müssen MHTML-Seiten dann i.a. eine Dateinamenserweiterung von „.html“ aufweisen.

```
<TITLE> Objekte, Klassen, Kapselung </TITLE>
```

Dieses Kapitel erlaeutert, wie Objekte beschrieben werden koennen,  
und geht dazu ausfuehrlich ... deutlich zu machen,  
werden Subtyping und Vererbung erst im naechsten Kapitel behandelt (  
siehe <A HREF="kap3.mhtml"> Kap. 3 </A> ).

```
<BR> <BR>
```

Abschnitte: <BR>

2.1 <A HREF="abs2.1.mhtml"> Objekte und Klassen </A>

```
<BR>
```

2.2 <A HREF="abs2.2.mhtml"> Kapselung und Strukturierung  
von Klassen </A>

Der Titel der W3-Seite muss am Anfang des Textes stehen und wie gezeigt geklammert sein (vgl. auch Abschn. 2.2.1, S. 112). Hypertext-Referenzen werden mit der Syntax

```
<A HREF="Dateiname"> Schaltflächenbeschriftung </A>
```

beschrieben. Der Dateiname gibt dabei an, wo die referenzierte Seite zu finden ist. In der Darstellung im Browser sollen Hypertext-Referenzen als Schaltflächen mit der angegebenen Beschriftung erscheinen. Beim Anklicken der Schaltfläche soll die referenzierte Seite geladen werden. Eine Oberfläche, die obige W3-Seite anzeigt, könnte wie in Abb. 5.20 aussehen.

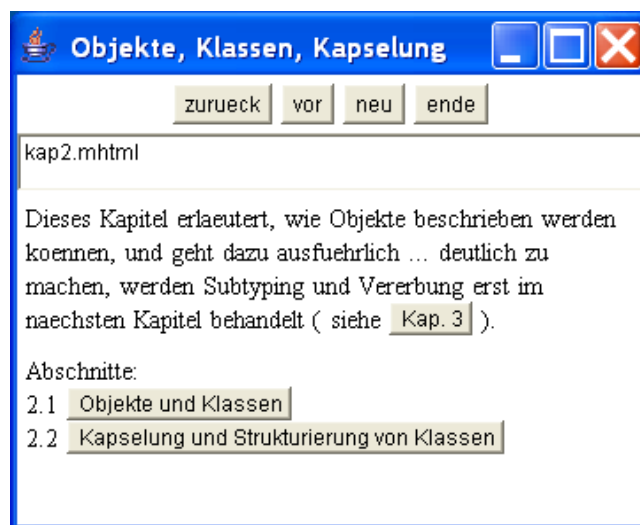


Abbildung 5.20: Browseroberfläche

Jeder Browser soll die betrachteten Seiten in einer vor-zurück-Liste verwalten (vgl. S. 223f) und dem Benutzer ein Vor- und Zurückgehen per Mausklick ermöglichen. Die Bedienoberfläche soll es darüber hinaus gestatten, weitere Browser zu öffnen und die gesamte Browseranwendung zu schließen (vgl. die Kommandos „neu“ und „ende“ von Abb. 2.7, S. 98).

### 5.3.2.2 Entwicklung von Anwendungsschnittstelle und Dialogführung

Die obige Anforderungsbeschreibung nennt fünf Browser-Operationen: Laden einer Seite, Vorgehen, Zurückgehen, Erzeugen eines neuen Browsers, Beenden der Anwendung. Betrachtet man die Benutzung eines Browsers als Dialog zwischen Benutzer und Browser, ergibt sich aus der Anforderungsbeschreibung eine Dialogführung gemäß Abb. 5.21.

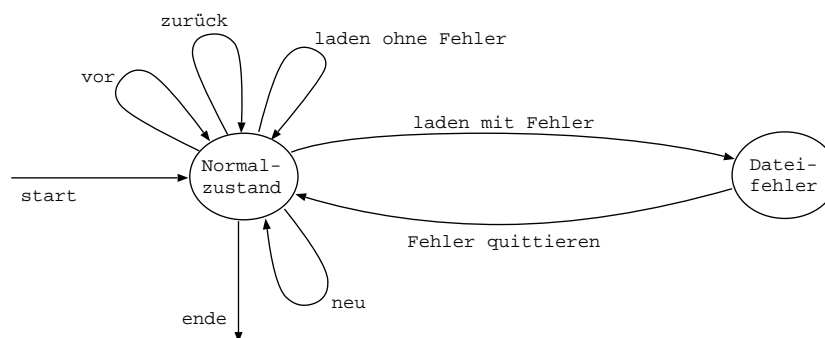


Abbildung 5.21: Dialogführung des Browsers

Nach dem Start befindet sich der Browser in einem Normalzustand, in dem alle Operationen möglich sind. Abgesehen von der Beende-Operation kehrt der Browser nach Ausführung einer Operation in diesen Normalzustand zurück. Kann er bei der Lade-Operation nicht auf die gesuchte Datei zugreifen, geht er in den Zustand „Dateifehler“. Durch Quittieren der Fehlermeldung bringt der Benutzer den Browser wieder in den Normalzustand. Man beachte, dass im Zustand „Dateifehler“ keine Browser-Operation zulässig sein soll.

Nachdem wir die Anforderungsbeschreibung soweit präzisiert haben, können wir uns an die Realisierung machen. Gemäß der MVC-Architektur wollen wir die Implementierung des Browsers von der Implementierung seiner Darstellung trennen. Aus Sicht von Anwendung und Dialogführung brauchen wir nur zu wissen, welche Darstellungen von der Anwendung benutzt werden sollen und wie der Zusammenhang zwischen Dialogführung und Darstellung sein soll. Die präzisierte Anforderungsbeschreibung legt folgenden Entwurf nahe: Im *Normalzustand* besteht die Darstellung aus einem Bedienfenster. Auftretende Änderungen meldet der Browser dem Bedienfenster durch die Nachricht „aktualisieren“. Im Zustand „Dateifehler“ wird ein modales Dialogfenster angezeigt, das das Bedienfenster solange blockiert, bis der Benutzer den Fehler quittiert hat (vgl. S. 339).

Die skizzierten Informationen über die Darstellung reichen für die Realisierung der Browser-Anwendung und der Dialogführung aus. Die Klasse

NetzSurfer in Abb. 5.22 liefert eine direkte Umsetzung des erläuterten Entwurfs. Sie stellt Methoden zum Laden, Vor- und Zurückgehen zur Verfügung. Mittels des Konstruktors können neue Browser erzeugt werden. Das Beenden aller Browser wird von der Bedienoberfläche realisiert.

Die Realisierung der Klasse `NetzSurfer` basiert auf den Browser-Implementierungen aus Kap. 2 und 3 (siehe S. 97, 223 und 225). Sie benutzt die Klasse `W3Seite` von Seite 112, `Hinweisfenster` aus Abb. 5.12, S. 339, für den Fehlerdialog, die Klasse `ExtendedList` aus Abschn. 3.3, S. 222, für die vor-zurück-Liste und die Klasse `DateiZugriff` von Seite 276, für den Zugriff auf die Webseiten. Die Anwendung der Klasse `DateiZugriff` ist auch insofern interessant, als sie eine typische Fehlerbehandlungssituation veranschaulicht: Der Fehler beim Dateiöffnen passiert in der Klasse `DateiZugriff`; da er dort aber nicht behandelt werden kann, wird er an die anwendende Klasse – hier `NetzSurfer` – weitergeleitet und von dieser dem Benutzer mitgeteilt. Bevor die Implementierung der Klasse `NetzSurfer` vorgestellt wird, werden die bereits an anderer Stelle eingeführten, referenzierten Implementierungsteile noch einmal im Zusammenhang aufgeführt. Die noch ausstehende Klasse `BedienFenster` behandeln wir in Unterabschnitt 5.3.2.3. `BedienFenster` besitzen eine Referenz auf die Browser-Anwendung, um die Operationen des Browsers aufrufen zu können.

```

/***** Klasse W3Seite *****/

class W3Seite {
    private String seite;

    W3Seite( String t, String i ) {
        seite = "<TITLE>" + t + "</TITLE>" + i ;
    }
    String getTitel(){
        int trennIndex = seite.indexOf("</TITLE>");
        return new String( seite.toCharArray(), 7, trennIndex-7 );
    }
    String getInhalt(){
        int trennIndex = seite.indexOf("</TITLE>");
        return new String( seite.toCharArray(), trennIndex+8,
                           seite.length() - (trennIndex+8) );
    }
}

/***** Klasse HinweisFenster *****/

import java.awt.*;

public class HinweisFenster extends Dialog {
    Label hinweisLabel;

    public HinweisFenster( Frame f, String hinw ) {
        super( f, true );    // true bedeutet modaler Dialog
    }
}

```

```

        hinweisLabel = new Label(hinw, Label.CENTER);
        add( hinweisLabel, BorderLayout.CENTER );

        Button ok = new DoButton("ok") {
            public void doAction() {
                HinweisFenster.this.setVisible(false);
            }
        };
        add( ok, BorderLayout.SOUTH );

        setSize( 200, 100 );
        setLocation( 400, 200 );
    }
    public void setText( String s ) {
        hinweisLabel.setText( s );
    }
}

/***** Klasse LinkedList *****/

import java.util.NoSuchElementException;

public class LinkedList {
    protected Entry header = new Entry(null, null, null);
    protected int size = 0;

    /* Constructs an empty Linked List. */
    public LinkedList() {
        header.next = header;
        header.previous = header;
    }
    /* Returns the last Element in this List. */
    public Object getLast() {
        if (size==0) throw new NoSuchElementException();
        return header.previous.element;
    }
    /* Removes and returns the last Element from this List. */
    public Object removeLast() {
        Entry lastentry = header.previous;
        if (lastentry == header) throw new NoSuchElementException();
        lastentry.previous.next = lastentry.next;
        lastentry.next.previous = lastentry.previous;
        size--;
        return lastentry.element;
    }
    /* Appends the given element to the end of this List. */
    public void addLast(Object e) {
        Entry newEntry = new Entry(e, header, header.previous);
        header.previous.next = newEntry;
        header.previous = newEntry;
        size++;
    }
}

```

```

/* Returns the number of elements in this List. */
public int size() {
    return size;
}

static class Entry {
    Object element;
    Entry next;
    Entry previous;

    Entry(Object element, Entry next, Entry previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
    }
}

public ListIterator listIterator() {
    return new ListIterator();
}

public class ListIterator {
    protected int nextIndex = 0;
    protected Entry next = header.next;

    public boolean hasNext() {
        return nextIndex != size;
    }

    public Object next() {
        if( nextIndex==size ) throw new NoSuchElementException();
        Object elem = next.element;
        next = next.next;
        nextIndex++;
        return elem;
    }
}

/***** Klasse ExtendedList *****/

import java.util.NoSuchElementException;

public class ExtendedList extends LinkedList {

    public ExtListIterator extListIterator() {
        return new ExtListIterator();
    }

    public class ExtListIterator extends ListIterator {

        public boolean hasPrevious() { return nextIndex != 0; }

        public Object previous() {

```

```

        if( nextIndex == 0 ) throw new NoSuchElementException();
        next = next.previous;
        nextIndex--;
        return next.element;
    }
    public void setToEnd() {
        next = header;
        nextIndex = size;
    }
    public void cut() {
        if( next != header ) {
            // es existieren zu entfernende Elemente
            header.previous = next.previous;
            next.previous.next = header;
            if( nextIndex == 0 ) header.next = header;
            size = nextIndex;
            next = header; // Iterator ans Ende setzen
        } // wenn next == header, ist nichts zu tun
    }
}
}

/***** Klasse DateiZugriff *****/

import java.io.*;

public class DateiZugriff {
    public static void schreiben( String dateiname, String s )
        throws IOException {
        PrintWriter out;
        out = new PrintWriter( new FileWriter( dateiname ) );
        out.print( s );
        out.close();
    }

    public static String lesen(String dateiname)
        throws FileNotFoundException, IOException {
        BufferedReader in =
            new BufferedReader(new FileReader(dateiname));
        StringBuffer inputstr = new StringBuffer("");
        String line;
        line = in.readLine();
        while (line != null) {
            inputstr = inputstr.append(line);
            inputstr = inputstr.append("\n");
            line = in.readLine();
        }
        in.close();
        return inputstr.toString();
    }
}

```

```

class NetzSurfer {
    protected BedienFenster oberfl;
    protected HinweisFenster fehlerDl;
    protected W3Seite         aktSeite;
    protected ExtendedList    vorzurueckliste;
    protected ExtendedList.ExtListIterator seitenIter;

    NetzSurfer() {
        aktSeite = new W3Seite(
            "Startseite", "NetzSurfer: Keiner ist kleiner");
        vorzurueckliste = new ExtendedList();
        vorzurueckliste.addLast( aktSeite );
        seitenIter = vorzurueckliste.extListIterator();
        seitenIter.setToEnd();
        oberfl      = new BedienFenster( this );
        oberfl.aktualisieren();
        fehlerDl=new HinweisFenster(oberfl,"Seite nicht gefunden");
    }
    void laden( String sadr ) {
        try {
            String titel, inhalt, s = DateiZugriff.read( sadr );
            int trennIndex = s.indexOf("</TITLE>");
            if( s.indexOf("<TITLE>") != 0 || trennIndex == -1 ){
                titel = "Ohne Titel"; // s hat keine Titeldeklaration
                inhalt = s;
            } else {
                titel = new String( s.toCharArray(),7,trennIndex-7 );
                inhalt = new String( s.toCharArray(), trennIndex+8,
                    s.length() - (trennIndex+8) );
            }
            aktSeite = new W3Seite( titel, inhalt );
            seitenIter.cut();
            vorzurueckliste.addLast( aktSeite );
            seitenIter.setToEnd();
            oberfl.aktualisieren();
        } catch( Exception e ) { fehlerDl.setVisible( true ); }
    }
    void vorgehen(){
        if( seitenIter.hasNext() ){
            aktSeite = (W3Seite)seitenIter.next();
            oberfl.aktualisieren();
        }
    }
    void zurueckgehen(){
        seitenIter.previous();
        if( seitenIter.hasPrevious() ){
            aktSeite = (W3Seite)seitenIter.previous();
            oberfl.aktualisieren();
        }
        seitenIter.next();
    }
}

```

Abbildung 5.22: Die Klasse NetzSurfer



### 5.3.2.3 Entwicklung der Darstellung

Grundsätzlich muss man bei der Entwicklung der Darstellung für jeden Dialogzustand eine graphische Präsentation entwerfen. Im betrachteten Beispiel wollen wir den Normalzustand durch ein Bedienfenster präsentieren, das eine W3-Seite anzeigt und es dem Benutzer ermöglicht, einen Dateinamen einzugeben und die Browseroperationen auszulösen. Im Zustand „Dateifehler“ soll zusätzlich ein Dialogfenster erscheinen. Das Aussehen des Dialogfensters ergibt sich durch die Verwendung der Klasse `HinweisFenster`. Für die Gestaltung der Bedienfenster orientieren wir uns an Abb. 5.20, S. 351. Eine mögliche Implementierung für Bedienfenster zeigt Abbildung 5.23. Alle Aspekte der Steuerung wurden dabei zunächst weggelassen.

Der Aufbau des Bedienfensters wird im Konstruktor vorgenommen. Die Schaltflächen werden einem Panel hinzugefügt. Dieses Panel wird zusammen mit dem Textfeld in ein Panel `bedienp` mit Grid-Layout eingetragen, das zwei Zeilen und eine Spalte besitzt. Da das Panel `bedienp` den oberen Teil des Bedienfensters ausmachen soll, wird es ihm im Norden hinzugefügt. Das Panel zur Darstellung der W3-Seite, kurz MHTML-Display genannt, füllt das Zentrum des Bedienfensters aus (vgl. auch Abb. 5.3, S. 312).

Für das Aktualisieren der W3-Seite ist die Methode `aktualisieren` zuständig. Sie übernimmt den Titel der aktuellen W3-Seite in die Titelzeile des Bedienfensters, entfernt alle Komponenten aus dem MHTML-Display und fügt mittels der Methode `addMhtmlComponents` die Komponenten hinzu, die dem Inhalt der aktuellen W3-Seite entsprechen. Schließlich wird die Bildschirmdarstellung aufgefrischt. (Auf Laufzeit-Optimierungen im Zusammenhang mit der Aktualisierung wurde hier bewusst verzichtet.)

Die Hauptarbeit beim Aktualisieren leistet die Hilfsmethode `addMhtmlComponents`. Sie zerlegt einen MHTML-Text in Token und fügt für jeden Token ein entsprechendes `BrMark`-, `HrefButton`- bzw. `WordComponent`-Objekt dem MHTML-Display hinzu, d.h. dem als Parameter übergebenen Panel (vgl. Abb. 5.24). Programmtechnisch ist nur die Zerlegung der Hypertext-Referenzen in ihre einzelnen Bestandteile etwas komplexer. Für die Realisierung der MHTML-Komponenten benutzen wir die in Unterabschn. 5.2.3.6 entwickelten Komponententypen `WordComponent` und `BrMark` sowie den `TextLayout-Manager`. Außerdem benötigen wir noch einen Komponententypen `HrefButton` zur Realisierung der Schaltflächen innerhalb des MHTML-Displays. Ein `HrefButton`-Objekt ist ein `Button`-Objekt, das zusätzlich ein Attribut `href` zur Speicherung des Dateinamens für die referenzierte Seite besitzt. Der Dateiname wird dem Konstruktor mitgegeben und kann mittels der Methode `getHref` ausgelesen werden. Eine mögliche Implementierung der Klasse `HrefButton` ist in Abb. 5.25 gezeigt.

```

class BedienFenster extends BaseFrame {
    String      text;
    NetzSurfer  surferAppl;
    Panel       mhtmlDisplay;

    public BedienFenster( NetzSurfer sa ) {
        surferAppl = sa;

        // Schaltflaechen
        Panel schaltp = new Panel();

        Button zursf = new DoButton( "zurueck" ){ .... }
        schaltp.add( zursf );
        Button vorsf = new DoButton( "vor" ){ .... }
        schaltp.add( vorsf );
        Button neusf = new DoButton( "neu" ){ .... }
        schaltp.add( neusf );
        Button endsf = new DoButton( "ende" ){ .... }
        schaltp.add( endsf );

        // Texteingabezeile
        TextField urlzeile = new TextField("",25);
        urlzeile.setEditable( true );
        ....
        Panel bedienp = new Panel( new GridLayout(2,1) );
        bedienp.add( schaltp );
        bedienp.add( urlzeile );

        // MHTML-Anzeige
        mhtmlDisplay = new Panel( new TextLayout() );
        mhtmlDisplay.setBackground( Color.white );

        add( bedienp, BorderLayout.NORTH );
        add( mhtmlDisplay, BorderLayout.CENTER );
    }

    void aktualisieren(){
        setTitle( surferAppl.aktSeite.getTitel() );
        text = surferAppl.aktSeite.getInhalt();

        mhtmlDisplay.removeAll();
        addMhtmlComponents( mhtmlDisplay, text );
        setVisible( true );
    }

    void addMhtmlComponents(Container p,String s) {
        ... // s. naechste Abbildung
    }
}

```

Abbildung 5.23: Realisierung der Darstellung von Bedienfenster

```

void addMhtmlComponents( Container p, String s ){
    StringTokenizer st = new StringTokenizer( s, " \n\r\t" );

    while( st.hasMoreTokens() ){
        String token = st.nextToken();

        if( token.equals("<BR>") ) {
            p.add( new BrMark() );

        } else if( token.equals("<A") && st.hasMoreTokens() ){
            token = st.nextToken();
            if( token.indexOf("HREF=\"") == 0
                && token.endsWith("\">") ) {

                // Entfernen von 'HREF="' und '>'
                String href = token.substring(6,token.length()-2);

                // Ermitteln der Schaltflaechenbeschriftung
                String aufKnopf = "";
                while( st.hasMoreTokens() ){
                    token = st.nextToken();
                    if( token.equals("</A>") ) break;
                    aufKnopf += ( " " + token );
                }
                // Entfernen des fuehrenden Leerzeichens
                aufKnopf.substring( 1 );

                // Hinzufuegen des HrefButtons und der Steuerung
                HrefButton hb = new HrefButton( aufKnopf, href );
                hb.addActionListener(
                    new ActionListener(){
                        public void actionPerformed((ActionEvent e) {
                            surferAppl.laden(
                                ((HrefButton)e.getSource()).getHref() );
                        }
                    } );
                p.add( hb );
            }
            // beachte: Wenn MHTML-Seite syntaktisch fehlerhaft,
            //          werden einige Zeichen verschluckt.

        } else { // d.h. weder BR-Markierung noch HREF-Item
            p.add( new WordComponent( token ) );
        }
    }
}

```

Abbildung 5.24: Methode zum Aufbau des MHTML-Displays

```

class HrefButton extends Button {
    String href;
    Dimension size;

    HrefButton( String l, String hr ){
        super( l );
        href = hr;
    }
    public String getHref() { return href; }

    public Dimension getMinimumSize() {
        return getPreferredSize();
    }
    public Dimension getPreferredSize() {
        Graphics g = getGraphics();
        if( size != null ) {
            return size;
        } else if( g == null ){
            return super.getPreferredSize();
        } else {
            Dimension d = super.getPreferredSize();
            FontMetrics fm = g.getFontMetrics();
            int textb = fm.stringWidth( getLabel() );
            int texth = fm.getHeight();
            size = new Dimension( textb+2*(d.width-textb)/3,
                                   texth+(d.height-texth)/2 );
            return size;
        }
    }
}

```

Abbildung 5.25: Schaltflächen in MHTML-Seiten

Um ein etwas schöneres Layout der MHTML-Seiten zu erzielen, ist der Rand der Schaltfläche etwas enger um deren Aufschrift gezogen. Dazu wurden die beiden größenbestimmenden Methoden `getPreferredSize` und `getMinimumSize` überschrieben.

#### 5.3.2.4 Realisierung der Steuerung

Der letzte Entwicklungsschritt besteht in der Realisierung der Steuerung. Sie stellt den operativen Zusammenhang zwischen Darstellung und Anwendung her. Dazu registrieren wir an den Schaltflächen und dem Textfeld der Darstellung Beobachter-Objekte. Diese sollen die entsprechenden Operationen im Browser auslösen. In unserem Beispiel lässt sich das leicht realisieren, da im Wesentlichen jeder Operation auf der Oberfläche eine Operation der Anwendung entspricht. Programmtechnisch geht es darum, die in Abb. 5.23 durch vier Punkte bezeichneten Lücken auszufüllen:

```

Button zursf = new DoButton( "zurueck" ){
    public void doAction() { surferAppl.zurueckgehen(); }
};
Button vorsf = new DoButton( "vor" ){
    public void doAction() { surferAppl.vorgehen(); }
};
Button neusf = new DoButton( "neu" ){
    public void doAction() { new NetzSurfer(); }
};
Button endsf = new DoButton( "ende" ){
    public void doAction() { System.exit( 0 ); }
};

TextField urlzeile = new TextField("",25);
urlzeile.setEditable( true );
urlzeile.addActionListener(
    new ActionListener(){
        public void actionPerformed((ActionEvent e) {
            surferAppl.laden( e.getActionCommand() );
        }
    }
);

```

Die Steuerung zu den Schaltflächen im MHTML-Display wurde bereits in Abb. 5.24 gezeigt.

### 5.3.2.5 Zusammenfassende Bemerkungen

Die skizzierte Entwicklungsmethode strukturiert eine GUI in drei Teile, wobei die Dialogführung der Anwendung zugerechnet wird. Die Dialogführung beschreibt das logische Verhalten der Bedienoberfläche, und zwar unabhängig von der Darstellung der Bedienoberfläche. Die Darstellung wird durch Spezialisierung von Komponenten des AWT realisiert. Dabei wird die gesamte Basisfunktionalität für GUIs geerbt. Die Anwendung besitzt eine Referenz auf ihre Darstellung bzw. – wenn es mehrere gibt – auf ihre Darstellungen. Sie benachrichtigt die Darstellungen über Änderungen und fordert sie damit zum Aktualisieren auf. Die Darstellungen besitzen Referenzen auf die Anwendung, um deren Daten auszulesen. Dies entspricht genau der MVC-Architektur.

Die Steuerung verbindet die Darstellung mit der Anwendung. Wir haben bei jeder aktiven Komponente der Darstellung genau ein Beobachter-Objekt registriert, das die entsprechenden Methoden der Anwendung aufruft. Mit dieser eindeutigen Entsprechung von Darstellungskomponenten und Steuerungselementen sind wir dem Architekturbild von Abb. 5.19 gefolgt. Dies führt – wie im Browser-Beispiel – oft dazu, dass die Steuerung programmtechnisch relativ eng an die Darstellung angebunden ist. Im Allgemei-

nen bietet das Beobachterkonzept, das der Steuerung im AWT zugrunde liegt, größere Flexibilität: Einerseits kann ein Steuerungselement mehrere Darstellungen beobachten, andererseits können bei einer Darstellungskomponente mehrere Steuerungen registriert sein.



## Selbsttestaufgaben

### Aufgabe 1: Addierer

In dieser Aufgabe wollen wir noch einmal die Programmierung mit Ereignissen und Fenstern am Beispiel erproben.

1. Beschreiben Sie dazu zunächst, was das folgende Programm bei seiner Ausführung macht.

```
import java.awt.*;
import java.awt.event.*;

class Addierer extends Frame {
    protected Button b1;
    protected TextField t1,t2,t3;

    public Addierer() {
        setLayout(new FlowLayout());
        setSize(300,100);
        b1 = new Button("addiere");
        t1 = new TextField("4",4);
        t2 = new TextField("2",4);
        t3 = new TextField(4);
        t3.setEditable(false);
        add(t1);
        add(t2);
        add(t3);
        add(b1);
    }
}

class AddiererTest {
    public static void main(String argv[]) {
        Addierer f = new Addierer();
        f.addWindowListener(new WindowAdapter () {
            public void windowClosing (WindowEvent e) {
                System.exit(0);
            }
        });
        f.setVisible(true);
    }
}
```

2. Erweitern Sie das Programm dann anschließend so, dass beim Drücken des Knopfes `b1` die Inhalte der Textfelder `t1` und `t2` in Integer-Zahlen umgewandelt, addiert und in das Textfeld `t3` geschrieben werden. Implementieren Sie dazu eine geeignete Beobachterklasse.

Geben Sie zwei Varianten für die Beobachter-Klasse an:



- (a) Die Beobachterklasse spezialisiert die Klasse `Addierer` so, dass sie als ihr eigener Beobachter für an `b1` auftretende Ereignisse verwendet werden kann.
- (b) Die Beobachterklasse wird als eine von `Addierer` unabhängige Klasse realisiert.

### Aufgabe 2: Ein minimaler Rechner

In dieser Aufgabe wollen wir uns noch einmal von den Vorzügen der Wiederverwendung durch Vererben von Programmteilen beim Programmieren im Kleinen überzeugen. Erweitern Sie dazu das Programm aus Aufgabe 1 um eine Möglichkeit zum Multiplizieren. Leiten Sie dazu von der Klasse `AddiererSpez` aus der Musterlösung zu dieser Aufgabe eine Klasse `MiniRechner` ab, deren Objekte zusätzlich einen Knopf mit der Aufschrift „multiplizieren“ zum Multiplizieren anzeigen. Registrieren Sie dann an diesem Knopf ein entsprechendes Beobachter-Objekt. Das neue Beobachter-Objekt soll die Methode `actionPerformed` der Klasse `AddiererSpez` wiederverwenden. Beide Knöpfe sollen dasselbe Beobachter-Objekt verwenden. Schreiben Sie zum Test Ihres Programms eine geeignete Testklasse.

### Aufgabe 3: Eine Zählerklasse

In dieser Aufgabe wollen wir die Programmierung mit Ereignissen noch einmal vertiefen und mit der Gestaltung von Benutzungsoberflächen verbinden. Dazu wollen wir eine Benutzungsoberfläche für die Applikation `Counter` entwickeln. Die Klasse `Counter` sieht dabei folgendermaßen aus:

```
class Counter {

    private int startValue;
    private int value;
    private int min = 0;
    private int max = 99;

    Counter (int start)    {
        startValue = start;
        value = start;
    }
    void incr() { if (value < max) value++; }
    void decr() { if (value > min) value--; }
    void resetCounter () {value = startValue;}
    public String toString() { return "" + value; }
}
```

Die in der Folge zu implementierende Benutzungsoberfläche soll wie in Abbildung 5.26 aussehen und drei Schaltflächen mit den Funktionalitäten *increment*, *decrement* und *quit* enthalten.



Abbildung 5.26: Eine Benutzungsoberfläche


Zusätzlich sind folgende Bedingungen einzuhalten:

- Das Hauptfenster in Abbildung 5.26 ist 120 x 100 Pixel groß und die oberen drei Schaltflächen befinden sich in einem Panel. Der Zählerstand steht dabei in einem nicht editierbaren Textfeld (vgl. die Methode `setEditable` in der Klasse `TextField`).
- Implementieren Sie zur Lösung der Aufgabe eine Klasse `CountFrame` als Subklasse von `Frame`.
- Registrieren Sie Beobachter-Objekte in geeigneter Weise, sodass der Zählerstand per Mausklick ereignis-gesteuert verändert werden kann. Bei Betätigung der Schaltfläche mit der Beschriftung „quit“ soll der Zähler auf den Anfangszustand zurückgesetzt werden.
- Das Programm soll beendet werden, wenn der Benutzer die Anforderung zum Schließen des Fensters gibt (unter Windows durch Anklicken der rechten oberen Schaltfläche ☒ des Fensters).
- Testen Sie Ihre Implementierung. Ergänzen Sie dazu Ihr Programm um eine Methode `main`, die ein Objekt vom Typ `CountFrame` erzeugt.



## Musterlösungen zu den Selbsttestaufgaben

### Aufgabe 1: Addierer

1. Die Klasse `Addierer` erzeugt ein Hauptfenster mit 3 Textfeldern und einem Button. Die Textfelder umfassen jeweils 4 Spalten. Der Text des ersten Textfeldes ist mit „4“ initialisiert, der des zweiten mit „2“ und der des dritten mit der leeren Zeichenkette. Der Button trägt die Beschriftung „addiere“. In den ersten beiden Textfeldern kann der Text beliebig geändert werden, im dritten nicht. Die Anwendung kann vom Benutzer durch die übliche Anforderung zum Schließen des Fensters beendet werden (unter Windows durch Anklicken der rechten oberen Schaltfläche  des Fensters).
2. Die folgenden zwei Beobachterklassen lösen die gestellte Aufgabe.

(a) Die Klasse `Addierer` kann wie folgt spezialisiert werden:

```
import java.awt.*;
import java.awt.event.*;

class AddiererSpez extends Addierer implements ActionListener {

    AddiererSpez () {
        b1.addActionListener(this);
    }
    public void actionPerformed (ActionEvent e) {
        try {
            Integer sum = new Integer(t1.getText()) +
                           new Integer(t2.getText());
            t3.setText(sum.toString());
        } catch (NumberFormatException ex) {
            System.out.println("Ungueltiger Eingabewert fuer t1 oder t2 : "
                               + ex.getMessage());
        }
    }
}

class AddiererTest {
    public static void main(String args[]) {
        Addierer f = new AddiererSpez();
        f.addWindowListener(new WindowAdapter () {
            public void windowClosing (WindowEvent e) {
                System.exit(0);
            }
        });
        f.setVisible(true);
    }
}
```

AddiererSpez **erweitert** Addierer und implementiert die Schnittstelle ActionListener. Im Konstruktor registriert sich das AddiererSpez-Objekt selbst als Beobachter von Action-Events an seinem Button b1. Wird der Button b1 gedrückt, wird der Rumpf der Methode actionPerformed ausgeführt, in dem die Inhalte beider Textfelder t1 und t2 addiert werden und das Ergebnis im dritten Textfeld (t3) ausgegeben wird.

- (b) In dieser Variante wird zur Realisierung von Beobachtern für b1 die Klasse AddierListener verwendet. Da sie nicht von Addierer erbt und daher nicht auf die Attribute t1, t2 und t3 der Klasse Addierer zugreifen kann, bekommt sie die entsprechenden Objekte im Konstruktor übergeben. Der ursprüngliche Konstruktor der Klasse Addierer wird erweitert um die Registrierung eines AddierListener-Objekts an b1.

```
class Addierer extends Frame {
    protected Button b1;
    protected TextField t1,t2,t3;

    public Addierer() {
        setLayout(new FlowLayout());
        setSize(300,100);
        b1 = new Button("addiere");
        t1 = new TextField("4",4);
        t2 = new TextField("2",4);
        t3 = new TextField(4);
        t3.setEditable(false);
        add(t1);
        add(t2);
        add(t3);
        add(b1);

        b1.addActionListener (new AddierListener (t1, t2, t3));
    }
}

class AddierListener implements ActionListener {

    private TextField t1, t2, t3;

    AddierListener (TextField t1, TextField t2, TextField t3) {
        this.t1 = t1;
        this.t2 = t2;
        this.t3 = t3;
    }

    public void actionPerformed (ActionEvent e) {
        try {
            Integer sum = new Integer(t1.getText()) +
                           new Integer(t2.getText());
            t3.setText(sum.toString());
        }
    }
}
```

```

        } catch (NumberFormatException ex) {
            System.out.println("Ungueltiger Eingabewert fuer t1 oder t2 : "
                               + ex.getMessage());
        }
    }
}

class AddiererTest {
    public static void main(String argv[]) {
        Addierer f = new Addierer();
        f.addWindowListener(new WindowAdapter () {
            public void windowClosing (WindowEvent e) {
                System.exit(0);
            }
        });
        f.setVisible(true);
    }
}

```

### Aufgabe 2: Ein minimaler Rechner

Der folgende Programmtext zeigt eine mögliche Lösung der Aufgabe.

```

class MiniRechner extends AddiererSpez {

    protected Button b2 = new Button("multipliziere");

    MiniRechner () {
        add (b2);
        b2.addActionListener(this);
    }
    public void actionPerformed (ActionEvent e) {
        if (e.getActionCommand().equals("addiere"))
            super.actionPerformed(e);
        else
            try {
                Integer prod = new Integer(t1.getText()) *
                               new Integer(t2.getText());
                t3.setText(prod.toString());
            } catch (NumberFormatException ex) {
                System.out.println("Ungueltiger Eingabewert fuer t1 oder t2 : "
                                   + ex.getMessage());
            }
    }
}

```

```
// Testprogramm
class MiniRechnerTest {
    public static void main(String argv[]) {
        MiniRechner f = new MiniRechner();
        f.addWindowListener(new WindowAdapter () {
            public void windowClosing (WindowEvent e) {
                System.exit(0);
            }
        });
        f.setVisible(true);
    }
}
```

Da die Klasse `MiniRechner` von `AddiererSpez` abgeleitet ist, erbt sie zusätzlich zur Funktionalität der Klasse `AddiererSpez` auch die Funktionalität der Klasse `Addierer`, die die direkte Superklasse von `AddiererSpez` ist. Da bei der Erzeugung eines `MiniRechner`-Objekts im Konstruktor der Klasse `MiniRechner` zunächst implizit der Default-Konstruktor von `AddiererSpez` aufgerufen wird, der selbst wiederum zunächst den Default-Konstruktor von `Addierer` aufruft, werden alle von den Superklassen benötigten Objekte generiert und bereits ein Beobachter-Objekt am Knopf `b1` registriert.

Die Methode `actionPerformed` der Klasse `MiniRechner` verwendet die der Klasse `AddiererSpez` durch Aufruf von `super.actionPerformed(e)`; wieder. Sie unterscheidet die Quelle des Ereignisses (`b1` oder `b2`) anhand der Beschriftung der Knöpfe (abfragbar durch `e.getActionCommand()`).

### Aufgabe 3: Eine Zählerklasse

Der folgende Programmcode löst die gestellte Aufgabe.

```
public class CountFrame extends Frame {

    Button bDec = new Button("<");
    Button bInc = new Button(">");
    Button bQuit = new Button("quit");
    TextField tCounter = new TextField(3);
    Counter c = new Counter(50);

    public CountFrame () {

        tCounter.setEditable(false);
        tCounter.setText(c.toString());

        // geeignete Beobachter registrieren
        bDec.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                c.decr();
            }
        });
    }
}
```

```
        tCounter.setText(c.toString());
    }
});
bInc.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        c.incr();
        tCounter.setText(c.toString());
    }
});
bQuit.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
        c.resetCounter();
        tCounter.setText(c.toString());
    }
});
addWindowListener (new WindowAdapter() {
    public void windowClosing (WindowEvent e) {
        System.exit(0);
    }
});

// Komponenten in richtiger Reihenfolge in Panel einfuegen
Panel p = new Panel(new FlowLayout());
p.add (bDec);
p.add (tCounter);
p.add (bInc);

// Komponenten in Hauptfenster einfuegen
add (p, BorderLayout.CENTER);
add (bQuit, BorderLayout.SOUTH);

// Fenstergroesse festlegen
setSize (120, 100);
}

public static void main(String[] args) {
    CountFrame countFrame = new CountFrame();
    countFrame.setVisible(true);
}
}
```





# Studierhinweise zur Kurseinheit 6

In dieser Kurseinheit werden Sie sich mit dem Kapitel 6 des Kurstexts beschäftigen. Die Kurseinheit führt Sie in die parallele Programmierung mit Threads ein. Sie sollten den gesamten Text zu dieser Kurseinheit studieren und verstehen. Prüfen Sie nach dem Durcharbeiten des Kurstextes, ob Sie die Lernziele erreicht haben. Dazu sollten Sie auch wieder die Aufgaben am Ende der Kurseinheit nutzen und durcharbeiten.

## **Lernziele:**

- Einordnung der verschiedenen Arten von Parallelität.
- Zusammenhang zwischen Parallelität und objektorientierter Programmierung.
- Detailliertes Verständnis des Threadkonzepts. (Was ist ein Thread; was ist ein Scheduler und welche Aufgabe hat er; welche Zustände kann ein Thread haben; wie kommunizieren Threads; wie unterscheiden sich Threads von Prozessen?).
- Umsetzung des Threadkonzepts in Java.
- Kenntnis der Anwendungsbereiche von Threads und der Gründe, warum sie in die Sprache Java integriert wurden.
- Kenntnis der Synchronisationsproblematik. (Was bedeutet Synchronisation; warum braucht man Synchronisationskonstrukte; welche Probleme können bei nicht vorhandener oder falsch realisierter Synchronisation auftreten?)
- Verständnis des Monitorkonzepts in der Form, wie es in Java realisiert ist.
- Verständnis des wait/notify-Mechanismus.



# Kapitel 6

## Parallelität in objektorientierten Programmen

Dieses Kapitel bietet im ersten Abschnitt eine allgemeine, kurze Einführung zu Parallelität in objektorientierten Programmen. Der zweite Abschnitt zeigt dann auf, welche Mittel Java für die parallele Programmierung bereitstellt. Dazu wird zunächst das Thread-Konzept erläutert. Anschließend werden Synchronisationsmechanismen behandelt.

### 6.1 Parallelität und Objektorientierung

Das Grundmodell der objektorientierten Programmierung bietet gute Voraussetzungen, um parallele Programmausführung zu unterstützen (vgl. Abschn. 1.2.3). Konzeptionell besteht die Ausführung eines objektorientierten Programms aus dem Verschicken von Nachrichten zwischen unabhängigen Objekten. Erhält ein Objekt eine Nachricht, führt es die entsprechende Methode aus und schickt ggf. eine Antwort. Parallelität bedeutet in diesem Grundmodell, dass möglicherweise mehrere Nachrichten gleichzeitig unterwegs sind und mehrere Objekte gleichzeitig Methoden ausführen.

In den zurückliegenden Kapiteln wurde diese Eigenschaft des Grundmodells nicht ausgenutzt: Alle Programme waren im Wesentlichen<sup>1</sup> sequentiell. Im objektorientierten Modell bedeutet sequentielle Ausführung, dass zu jedem Zeitpunkt genau ein Objekt aktiv ist; verschickt dieses Objekt eine Nachricht an ein anderes Objekt, wird es inaktiv und das Empfängerobjekt setzt die Programmausführung fort. Erst wenn das Empfängerobjekt die entsprechende Methode ausgeführt und geantwortet hat, wird das Senderobjekt wieder aktiv.

Im Folgenden gehen wir zunächst kurz auf einige allgemeine Aspekte von Parallelität ein, um einen Rahmen für die anschließenden Ausführungen zu

---

<sup>1</sup>Das AWT nutzt intern Parallelität.

haben, und erörtern dann Möglichkeiten und Probleme bei der Realisierung von Parallelität in objektorientierten Sprachen.

### 6.1.1 Allgemeine Aspekte von Parallelität

In vielen Softwaresystemen können bestimmte Systemteile prinzipiell parallel, d.h. zeitgleich ausgeführt werden. In der Praxis kommen in solchen Fällen folgende Konstellationen zum Einsatz:

1. Die Systemteile sind auf *mehrere* Rechner verteilt und kommunizieren über Netzwerke miteinander. Beispiele: Buchungssysteme (etwa im Luftverkehr oder bei der Bahn), verteilte Datenbanken, WWW.
2. Jedes Systemteil läuft in einem *eigenen* Prozess ab, aber alle Prozesse des Softwaresystems werden auf *einem* gemeinsamen Rechner ausgeführt. Jeder Prozess hat seinen eigenen Speicher und kommuniziert mit anderen Prozessen über Nachrichten/Signale, Dateien und Puffer.
3. Das Softwaresystem wird im Rahmen *eines* Prozesses ausgeführt. Meistens ist das System dann durch ein Programm in einer Programmiersprache beschrieben. Die Teile des Softwaresystems kommunizieren in erster Linie über den gemeinsamen Speicher.

Die ersten beiden Fälle weisen viele gemeinsame Eigenschaften auf. Die Grenze zwischen diesen Fällen ist in der Praxis oft fließend. Deshalb werden sie üblicherweise unter dem Begriff „verteilte Systeme“ subsumiert. Programmieraspekte verteilter Systeme behandeln wir in Kap. 7. In diesem Kapitel konzentrieren wir uns auf *lokale* Parallelität, d.h. Parallelität im Rahmen eines Prozesses.



*lokale  
Parallelität*

*reale  
Parallelität*

Systeme, bei denen Teile prinzipiell parallel ausgeführt werden können, können *real parallel* oder *virtuell parallel* ausgeführt werden. *Reale Parallelität* in einem System bedeutet, dass es Zeitpunkte gibt, an denen mehrere Aktionen exakt zeitgleich ablaufen. *Virtuelle<sup>2</sup> Parallelität* bedeutet, dass die Aktionen eines Systems, die im Prinzip zeitgleich ausgeführt werden könnten, in irgendeiner Art und Weise sequenzialisiert und nacheinander ausgeführt werden müssen. Dies ist z.B. dann der Fall, wenn statt eines Mehrprozessor-Systems nur ein Einprozessor-System zur Verfügung steht. Trotz der erzwungenen Sequenzialisierung ist bei virtueller Parallelität die Ausführungsreihenfolge der Aktionen im System nicht vollständig festgelegt. Sind zum Beispiel *a* und *b* zwei Aktionen, deren Ausführungsreihenfolge nicht festgelegt ist (sie könnten in einem real parallelen System zeitgleich ausgeführt werden), kann das ausführende System zuerst *a* und dann *b* ausführen oder erst *b* und dann

*virtuelle  
Parallelität*

<sup>2</sup>Virtuelle Parallelität wird oft auch als Quasi-Parallelität bezeichnet; wir benutzen hier das Begriffspaar real/virtuell in Anlehnung an viele ähnliche Situationen in der Informatik, wie z.B. reales/virtuelles Betriebssystem, realer/virtueller Speicher.

a. Auf die Umsetzung virtueller Parallelität in konkreten Rechensystemen können wir hier nur am Rande eingehen.

Im Vergleich zu einem streng sequentiellen Programmiermodell bringt bereits virtuelle Parallelität viele Vorteile mit sich. Sie erlaubt eine natürlichere Modellierung von softwaretechnischen Aufgabenstellungen, die von vornherein Parallelität aufweisen (beispielsweise Simulationsaufgaben, vgl. Abschn. 1.1.2), und vermeidet damit eine explizite, meist künstliche Sequentialisierung der Aktionen bei der Beschreibung derartiger Aufgabenstellungen. Außerdem bietet sie dem ausführenden Rechensystem die Möglichkeit, parallele Teilaufgaben verschränkt auszuführen, d.h. es werden abwechselnd Aktionen für die verschiedenen Teilaufgaben ausgeführt. Zum Beispiel kann man beim Laden einer Webseite den Aufbau der Bilder mit der Ausgabe des Texts verschränken, sodass die Textausgabe nicht warten muss, bis Bilder vollständig geladen sind.

**Ausführungsstränge** Die Ausführung eines sequentiellen Programms ist eine Folge von elementaren Aktionen. Eine Folge von solchen Aktionen nennen wir einen *Ausführungsstrang* (englisch *execution thread*). Zu jeder Aktion gibt es eine Programmstelle (eine Anweisung bzw. einen Teil einer Anweisung), die diese Aktion beschreibt. Einen Ausführungsstrang kann man sich also als den Weg vorstellen, den die Ausführung durch den Programmtext nimmt. In einem parallelen Programm können mehrere Ausführungsstränge nebeneinander existieren. Dadurch treten Operationen und Fragestellungen auf, die es in der sequentiellen Programmierung nicht gibt:

*Ausführungs-  
strang*

1. Erzeugen und Starten zusätzlicher Ausführungsstränge.
2. Synchronisieren mehrerer Ausführungsstränge.
3. Steuern der Ausführung, d.h. steuern welcher Strang wann und wie lange rechnen darf.

Diese drei Punkte sollen kurz erläutert werden. Üblicherweise wird auch ein paralleles Programm mit einem Ausführungsstrang gestartet. Es kann dann zur Laufzeit weitere Stränge erzeugen; d.h. ein Ausführungsstrang kann einen weiteren Strang starten. Dafür müssen die Programmiersprache und das Laufzeitsystem entsprechende Operationen bereitstellen. Die Programmausführung terminiert, wenn alle ihre Stränge terminieren.

Die Synchronisation hat im Wesentlichen zwei Aufgaben:

1. Sie soll sicherstellen, dass sich in bestimmten, sogenannten *kritischen* Programmbereichen maximal ein Ausführungsstrang befindet. (Kritische Bereiche sind i.a. Bereiche in denen von verschiedenen Ausführungssträngen gemeinsame Daten manipuliert werden. Kritische Bereiche werden auf Seite 400 näher erläutert.)

2. Sie soll es ermöglichen, die Tätigkeiten unterschiedlicher Ausführungsstränge zu koordinieren. Beispielsweise muss ein Ausführungsstrang, der einen Verbraucher implementiert, mit dem entsprechenden Erzeugerstrang koordiniert werden, da er nur verbrauchen kann, was vorher erzeugt wurde.

Scheduler

Im Allgemeinen ist es nicht möglich, jedem Ausführungsstrang einen eigenen Prozessor zuzuteilen. Folglich können die Stränge nicht real parallel ausgeführt werden. Die Steuerung, welcher Strang, wann und wie lange rechnen darf, d.h. die Zuteilung der vorhandenen Prozessoren an die Ausführungsstränge wird vom sogenannten *Scheduler*<sup>3</sup> vorgenommen. Man unterscheidet *preemptives* und *nicht-preemptives* Scheduling.

*fares*  
Scheduling

Beim preemptiven Scheduling entscheidet der Scheduler darüber, wie lange ein Ausführungsstrang rechnen kann; beispielsweise kann der Scheduler einem rechnenden Strang den Prozessor nach Überschreiten einer Zeitschranke entziehen und einem anderen Strang zuteilen. Garantiert der Scheduler, dass jeder rechenbereite Strang nach endlicher Zeit einen Prozessor erhält, spricht man von *fairem* Scheduling.

Beim nicht-preemptiven Scheduling entscheidet der rechnende Ausführungsstrang darüber, wann er den Prozessor freigibt. Gibt es nur einen Prozessor, kann er also alle anderen Stränge blockieren.

Während beim preemptiven Scheduling der Scheduler, d.h. das zugrunde liegende Laufzeit- bzw. Betriebssystem, für faires Scheduling sorgen kann, muss sich beim nicht-preemptiven Scheduling der Programmierer darum kümmern, dass jeder Ausführungsstrang die Möglichkeit zum Rechnen erhält.

atomare Aktion

Die Scheduling-Strategie kann das Verhalten von parallelen Programmen wesentlich beeinflussen. Dabei spielt nicht nur der Unterschied zwischen preemptiv und nicht-preemptiv eine Rolle. Wichtig ist auch zu wissen, an welchen Stellen ein Programm vom Scheduler unterbrochen werden kann, was *atomare*, d.h. nicht-unterbrechbare Aktionen sind, ob der Scheduler die Ausführungsstränge fair behandelt und inwieweit er Prioritäten zwischen den Strängen berücksichtigt. Entwickelt man Programme, die auf verschiedenen Rechensystemen mit unterschiedlichen Scheduling-Strategien ausgeführt werden sollen, muss man sehr sorgfältig darauf achten, dass durch geeignete Synchronisations- und Koordinationsmechanismen die Freiheitsgrade der Scheduler so eingeschränkt werden, dass das Programm unabhängig von der jeweiligen Scheduling-Strategie das beabsichtigte Verhalten zeigt.

---

<sup>3</sup>Aussprache: skedjuler.

### 6.1.2 Parallelität in objektorientierten Sprachen

Parallelität kann in objektorientierten Sprachen auf unterschiedliche Weise realisiert werden. Wir betrachten hier die zwei wichtigsten Varianten:

1. Ausführungsstränge wechseln zwischen Objekten: Dieses ist der Fall, den wir in den vorangegangenen Kapiteln betrachtet haben. (Da wir dort nur sequentielle Programme verwendet haben, gab es nur einen einzigen Ausführungsstrang<sup>4</sup>.) Trifft ein Ausführungsstrang auf einen Methodenaufruf, wechselt er vom Sender- zum Empfängerobjekt und kehrt erst nach Beendigung der Methode zurück. Damit man Ausführungsstränge ansprechen kann, werden sie in der objektorientierten Programmierung meist durch spezielle Objekte repräsentiert, denen man Nachrichten schicken kann, beispielsweise um die Ausführung zu starten, zu unterbrechen oder zu beenden (siehe unten).
2. Jedes Objekt hat seinen eigenen Ausführungsstrang: Dieser Ausführungsstrang führt lokale Berechnungen aus und bearbeitet eintreffende Nachrichten. Treffen mehrere Nachrichten gleichzeitig ein bzw. trifft eine Nachricht ein, während der Ausführungsstrang noch beschäftigt ist, muss die Nachricht gepuffert werden. Dementsprechend muss der Ausführungsstrang einen impliziten oder expliziten Mechanismus besitzen, der prüft, ob Nachrichten im Puffer vorhanden sind.

Die beiden Varianten unterscheiden sich hauptsächlich im Kommunikations- und Synchronisationsverhalten. In der ersten Variante können sich gleichzeitig mehrere Ausführungsstränge in den Methoden zu einem Objekt befinden; d.h. insbesondere, dass die Attribute gleichzeitig von mehreren Ausführungssträngen gelesen und modifiziert werden können, was leicht zu inkonsistenten Attributbelegungen führen kann. Andererseits ist die erste Variante eine einfache Erweiterung der sequentiellen Ausführung und damit leichter zu handhaben.

In der zweiten Variante haben die Objekte die Kontrolle über die Ausführung, sodass die skizzierten Inkonsistenzen leichter vermieden werden können. Dafür müssen sich die Objekte, d.h. der Programmierer, explizit um die Kommunikation mit anderen Objekten kümmern – beispielsweise durch regelmäßiges Abfragen der Nachrichtenpuffer. Andererseits ermöglicht diese Variante auch allgemeineres Kommunikationsverhalten, wie zum Beispiel das Arbeiten mit Nachrichten, die gleichzeitig an alle Objekte (broadcast) oder mehrere andere Objekte (multicast) verschickt werden. Außerdem bietet sie eine geeignetere Modellierung für die Programmierung verteilter Systeme (vgl. Kap. 7).

---

<sup>4</sup>Ausnahme: AWT



## 6.2 Lokale Parallelität in Java-Programmen

Im vorangegangenen Abschnitt wurden zwei Varianten zur Realisierung von Parallelität in objektorientierten Programmen erläutert. Für prozesslokale Parallelität stützt sich Java auf das Konzept der ersten Variante. Es stellt eine Thread-Bibliothek und sogenannte Objektmonitore für die Synchronisation zur Verfügung. Im Folgenden beschreiben wir zunächst die wesentlichen Eigenschaften der Java-Threads und behandeln dann die Synchronisationsmechanismen. Abschließend fassen wir die sprachliche Realisierung von lokaler Parallelität zusammen und diskutieren, inwieweit dabei objektorientierte Konzepte berücksichtigt wurden.

### 6.2.1 Java-Threads

Dieser Abschnitt erläutert zunächst, wie Ausführungsstränge programmtechnisch realisiert sind, welche Zustände sie annehmen und wie sie gesteuert werden können. Dann demonstriert er deren Anwendung und weitere Eigenschaften an einem kleinen Beispiel.

#### 6.2.1.1 Programmtechnische Realisierung von Threads in Java

Jeder Ausführungsstrang wird in Java durch ein Objekt des Typs `Thread` repräsentiert; d.h. das repräsentierende Objekt ist entweder ein Objekt der vordefinierten Klasse `Thread` oder eines ihrer Subklassen. An das `Thread`-Objekt werden die Nachrichten geschickt, mit denen der Ausführungsstrang gesteuert wird. Im Folgenden werden wir einen Ausführungsstrang mit dem `Thread`-Objekt identifizieren, das ihn repräsentiert, und einfach nur von *Threads* sprechen. Um Threads einzusetzen, muss man wissen, wie man die von einem Thread auszuführenden Aktionen beschreiben kann, wie man Threads starten und steuern kann und wie sich Threads verhalten.

*Thread*

**Beschreiben von Threads.** Die Aktionen, die ein Thread ausführen soll, werden in einer parameterlosen Methode mit Namen `run` beschrieben. Selbstverständlich können von der `run`-Methode weitere Methoden aufgerufen werden. Es gibt in Java zwei Möglichkeiten, diese `run`-Methode zu deklarieren. Entweder man leitet eine neue Klasse von der Klasse `Thread` ab und überschreibt deren `run`-Methode oder man implementiert mit einer Klasse die Schnittstelle `Runnable`:

```
interface Runnable { void run(); }
```

Das Muster zur Anwendung der ersten Möglichkeit sieht wie folgt aus:

```

class MeinThread1 extends Thread {
    ...
    public void run() {
        // Hier steht der Programmtext, der von Threads des
        // Typs MeinThread1 ausgeführt werden soll.
    }
}

```

Die Ausführung eines Threads vom Typ `MeinThread1` wird gestartet, indem man ein `MeinThread1`-Objekt erzeugt und für dieses Objekt die Methode `start` aufruft<sup>5</sup>:

```

Thread meinThread = new MeinThread1();
...
meinThread.start();

```

Das Laufzeitsystem von Java gibt der `start`-Methode eine spezielle Semantik: Sie stößt die Ausführung der Methode `run` des Threads an und kehrt sofort zum aufrufenden Programmkontext zurück, ohne die Terminierung des `run`-Aufrufs abzuwarten. Nach Rückkehr von `start` befinden sich also sowohl der Thread, in dem `start` aufgerufen wurde, als auch der neu angestoßene Thread in Ausführung.

Die zweite Möglichkeit, um Ausführungsstränge einzuführen, ist Folgende. Die Klasse, die einen Ausführungsstrang implementieren soll, erbt nicht von der Klasse `Thread`, sondern implementiert statt dessen die Schnittstelle `Runnable`. Dadurch stellt diese Klasse ihre Implementierung der `run`-Methode zur Verfügung:

```

class MeinThread2 extends WichtigeSuperklasse
                    implements Runnable {
    ...
    public void run() {
        // Hier steht der Programmtext, der den
        // Ausführungsstrang implementiert.
        // Die Implementierung kann Attribute und Methoden
        // der Klasse WichtigeSuperklasse benutzen.
    }
}

```

Um Ausführungsstränge starten zu können, die festgelegt sind durch die `run`-Methode einer Klasse *K*, die `Runnable` implementiert und nicht von `Thread` abgeleitet ist, muss man wie folgt vorgehen:

---

<sup>5</sup>Als Typ der Variablen `meinThread` hätte man auch `meinThread1` wählen können. Die aktuelle Deklaration hilft uns, diese Realisierung mit der über die Implementierung von `Runnable` besser vergleichen zu können.

1. Man muss zunächst ein Objekt *o* der Klasse *K* erzeugen. In unserem Beispiel ist das ein *MeinThread2*-Objekt.
2. Dann muss man ein neues Objekt *t* der Klasse *Thread* erzeugen.
3. Bei der Erzeugung ist dem Konstruktor das Objekt *o* zu übergeben.
4. Auf dem *Thread*-Objekt *t* wird die Methode `start` aufgerufen.

Nun startet das *Thread*-Objekt *t* die `run`-Methode des Objekts *o*, das sie ja durch die Übergabe im Konstruktor kennt. *t* ist quasi das Stellvertreter-Objekt für *o*. Sämtliche Methodenaufrufe, die zur Steuerung des Ausführungsstrangs nötig sind, müssen auf *t* erfolgen und nicht auf *o*.

Das folgende Beispiel demonstriert die Erzeugung eines Ausführungsstrangs für Objekte vom Typ *MeinThread2*. Dabei heißt *o* *meinTarget* und *t* *meinThread*.

```
Runnable  meinTarget = new MeinThread2();
Thread    meinThread = new Thread( meinTarget );
...
meinThread.start();
```

Dass *meinThread* die `run`-Methode des Objekts *meinTarget* aufruft, kann man verifizieren, indem man die `run`-Methode der vordefinierten Klasse *Thread* betrachtet:

```
public void run() {
    if( target != null ) target.run();
}
```

Durch den Konstruktoraufruf mit dem Argument `meinTarget` wird das Attribut `target` mit `meinTarget` initialisiert. Die `run`-Methode der Klasse *Thread* ruft demzufolge die `run`-Methode des Targets auf.

Während die erste Variante zur Beschreibung von Threads knapper und ein wenig einfacher ist, ermöglicht die zweite Variante, Threads zu entwickeln, die von anderen Klassen erben (im Beispiel die Klasse *WichtigeSuperklasse*). Insbesondere können auf diese Weise existierende Klassen mit der Fähigkeit ausgestattet werden, Ausführungsstränge zu definieren, die parallel zu anderen ausgeführt werden können. Wir sagen dann auch kürzer, dass solche Klassen als Threads ausführbar sind.

**Ablauf- und Zustandsverhalten von Threads.** Grundsätzlich kann sich ein Thread in vier verschiedenen Zuständen befinden: Er kann *neu* sein; d.h. das Thread-Objekt wurde erzeugt, aber die Methode `start` noch nicht aufgerufen. Er kann *lauffähig* sein; d.h. er wird gerade ausgeführt oder könnte ausgeführt werden, sofern ihm ein Prozessor zugeteilt würde. Seine Ausführung kann *blockiert* sein; d.h. er befindet sich in einem Zustand, in dem ein externes Ereignis abgewartet wird. Er kann *tot* sein; d.h. die Ausführung seiner `run`-Methode hat terminiert.

Für ein genaueres Verständnis des Ablauf- und Zustandsverhaltens von Threads ist es hilfreich, den Zustand „lauffähig“ in die Zustände „rechenbereit“ und „rechnend“ zu trennen sowie den Zustand „blockiert“ gemäß den Gründen für die Blockierung in verschiedene Zustände aufzuteilen. Abbildung 6.1 zeigt das resultierende Zustandsübergangsdiagramm. In jedem Zustand können sich im Prinzip beliebig viele Threads befinden; nur im Zustand „rechnend“ ist die Anzahl der Threads durch die Anzahl der Prozessoren begrenzt, die dem Prozess, in dem die Threads ablaufen, zugeteilt sind. Im Folgenden geben wir eine kurze Erläuterung des Zustandsübergangsdiagramms. Die Zustände „wartend“ und „Monitor blockiert“ werden genauer in Abschn. 6.2.2 beschrieben.

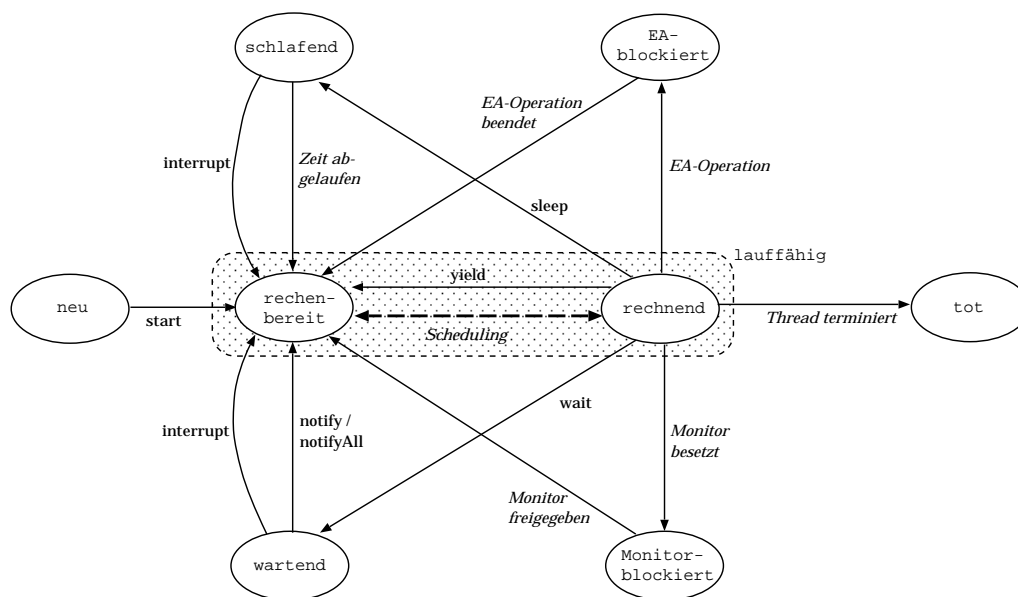


Abbildung 6.1: Zustandsverhalten von Threads

Threads im Zustand „rechenbereit“ könnten rechnen und warten darauf, dass ihnen der Scheduler einen Prozessor zuteilt. Threads im Zustand „rechnend“ befinden sich in Ausführung auf einem Prozessor. Mittels der statischen Methode `yield` kann ein rechnender Thread *t* dem Scheduler mitteilen, dass er willens ist, den Prozessor frei zu geben. In der Regel wird

der Scheduler dann den Thread  $t$  in den Zustand „rechenbereit“ versetzen und unter Berücksichtigung der Thread-Prioritäten einen der rechenbereiten Threads rechnen lassen (ggf. kann dies wieder  $t$  sein). Der Scheduler kann auch von sich aus einen rechnenden Thread unterbrechen und damit in den Zustand „rechenbereit“ versetzen. Die Semantik von Java schreibt aber nicht vor, ob und wann der Scheduler einen rechnenden Thread unterbricht.

In Java gibt es vier Zustände, in denen ein Thread blockiert ist<sup>6</sup>:

1. schlafend: Durch Aufruf der statischen Methode `sleep` kann sich ein Thread für eine Zeit schlafen legen. Ist die Zeit abgelaufen, wird er wieder rechenbereit.
2. EA-blockiert: Solange ein Thread auf den Abschluss einer Ein- oder Ausgabeoperation wartet, ist er blockiert; d.h. andere Threads können in der Zeit rechnen.
3. Monitor-blockiert: Ein Thread, der einen besetzten Monitor betreten will, ist solange blockiert, bis der Monitor frei gegeben wird und er ausgewählt wird, den Monitor zu betreten. (Was Monitore genau sind und wozu sie eingesetzt werden wird in Abschn. 6.2.2 besprochen. Hier genügt es zunächst zu wissen, dass ein Monitor eine Überwachungseinheit ist, die den Zugang von Threads zu bestimmten Anweisungsfolgen kontrolliert. Betritt ein Thread einen Monitors  $M$ , kann er von  $M$  überwachte Anweisungsfolgen ausführen und sperrt die Ausführung für andere Threads.)
4. wartend: Ein Thread, der sich in einem Monitor  $M$  befindet, d.h. der von einem Monitor  $M$  überwachte Anweisungen ausführt, kann seine Ausführung mittels der Methode `wait` aussetzen. Er wird in die Wartemenge von  $M$  eingereiht, und  $M$  wird frei gegeben. Dadurch können andere Threads, die auf die Ausführung von durch  $M$  überwachte Anweisungsfolgen warten,  $M$  betreten und die gewünschten Anweisungen ausführen. Mittels der Methoden `notify` und `notifyAll` werden wartende Threads wieder rechenbereit.

Man beachte die unterschiedlichen Signaturen der erwähnten Methoden: `sleep` und `yield` sind statische Methoden der Klasse `Thread`; sie beziehen sich immer auf den Thread, der ihren Aufruf ausführt. Die Methode `start` ist nicht statisch, d.h. sie hat einen impliziten Parameter vom Typ `Thread`. Die Methoden `notify`, `notifyAll` und `wait` sind in der Klasse `Object` deklariert und werden von dort an alle Objekte vererbt. Sie beziehen sich auf einen Monitor, den der ausführende Thread betreten hat.

---

<sup>6</sup>In einer älteren Version von Java gab es zusätzlich noch einen Zustand „suspendiert“.

### 6.2.1.2 Benutzung von Threads

Nachdem wir die allgemeine Funktionsweise von Threads erläutert haben, wollen wir ihre Benutzung und weitere Eigenschaften an einem kleinen Beispiel demonstrieren. Dazu betrachten wir zunächst eine modifizierte Variante vom sogenannten Spiel des Lebens und diskutieren dann Aspekte der Kommunikation und des Scheduling von Threads.

**Spiel des Lebens.** Das Spiel des Lebens simuliert in rudimentärer Weise die Entwicklung einer Population von Lebewesen (es wurde von John Conway entwickelt und heißt in seiner Originalfassung „Game of Life“). Die Welt besteht dabei aus einer endlichen Menge von Feldelementen. Feldelemente werden durch zwei natürlichzahlige Koordinaten bezeichnet, beispielsweise  $(lg, bg)$ . Die erste Koordinate nennen wir den Längengrad, die zweite den Breitengrad. Dabei gehen wir davon aus, dass die Gradzahlen von 0 bis  $MXG - 1$  laufen ( $MXG$  steht für **m**aximale **G**radzahl). Jedes Feldelement hat acht Nachbarorte; beispielsweise ist  $(8, 13)$  ein Nachbarort von  $(8, 12)$  und  $(0, 7)$  ein Nachbarort von  $(MXG - 1, 8)$ . Jedes Feldelement ist entweder von einem Lebewesen bewohnt oder leer.

Abbildung 6.2 zeigt ein mögliches Aussehen einer solchen Oberfläche mit  $MXG = 20$ , wobei Lebewesen in Gelb und leere Felder in Blau dargestellt sind.

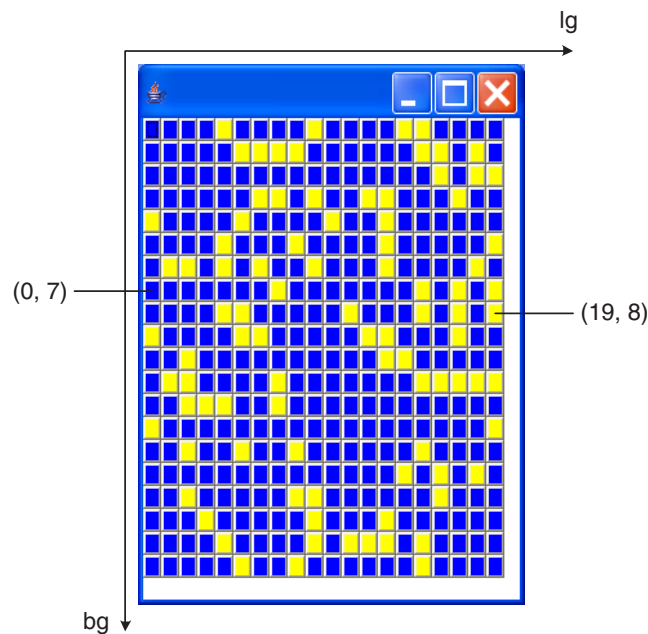


Abbildung 6.2: Einfache Oberfläche für die „Welt“

Die Entwicklung der Population vollzieht sich nach den folgenden nicht-deterministischen Regeln:

- Jedes Lebewesen lebt eine zufällig bestimmte Zeitspanne vor sich hin und sucht dann den Kontakt zu den Lebewesen der Nachbarorte: Hat es weniger als zwei Nachbarn, stirbt es an Vereinsamung; hat es mehr als drei Nachbarn, stirbt es an Überbevölkerung; andernfalls lebt es weiter.
- Bei leeren Feldelementen wird nach zufällig bestimmten Zeitspannen ermittelt, ob dort neues Leben entstehen kann. Dies ist der Fall, wenn das Feldelement genau drei Lebewesen in der Nachbarschaft hat.

Jedes Lebewesen wollen wir durch einen Thread realisieren, der die Nachbarschaft beobachtet und gegebenenfalls terminiert. Bei leeren Feldelementen verfahren wir ganz analog: Zu deren Beobachtung sehen wir ein fiktives Wesen vor, das wir Leerwesen nennen und das gemäß obiger Regel neues Leben erzeugt. Die gemeinsamen Eigenschaften von Lebe- und Leerwesen beschreiben wir in der Klasse `Wesen` (siehe Abb. 6.3).

Jedes Wesen ist ein Thread, der eine Referenz auf die Welt besitzt und seine Koordinaten kennt. Die Wesen führen ein recht eintöniges Dasein: Sie prüfen von Zeit zu Zeit, ob sie handeln müssen; ansonsten schlafen sie. Lebewesen müssen handeln, wenn die Anzahl ihrer Nachbarn ungleich zwei oder drei ist. Sie erzeugen dann ein Leerwesen, setzen es auf ihr Feldelement in die Welt, starten es und hauchen ihr Leben aus (siehe Abb. 6.3). Leerwesen handeln, wenn die Anzahl der Nachbarn gleich drei ist. Dann ersetzen sie sich durch ein Lebewesen.

Abbildung 6.4 zeigt eine mögliche Implementierung für die Welt. Das Weltfeld enthält die Feldelemente. Wir gehen davon aus, dass die Welt eine geeignete graphische Oberfläche besitzt wie in Abbildung 6.2 bereits gezeigt. Von Interesse für die Benutzung von Threads ist das Vorgehen im Konstruktor zur Erschaffung der Welt: Er erzeugt zunächst alle Wesen (mit einer Wahrscheinlichkeit von 0.25 werden Lebewesen geschaffen). Dann legt er die Oberfläche an und erst zum Schluss werden die Wesen gestartet. Dies ist ein gutes Beispiel, um zu verstehen, warum ein Thread nicht automatisch nach der Erzeugung gestartet wird. Wären die Wesen gleich nach der Erzeugung gestartet worden, hätten sie möglicherweise beim Ermitteln der Anzahl ihrer Nachbarn auf ein nicht-initialisiertes Feldelement zugegriffen, d.h. für dieses Feldelement wäre noch kein Lebe- oder Leerwesen erzeugt worden. Ein solcher Zugriff hätte eine `NullPointerException` ausgelöst. Die Trennung von Thread-Erzeugung und Thread-Starten ist also nötig, um beim Arbeiten mit mehreren Threads vor dem Starten einen konsistenten Programmzustand herzustellen.

```

abstract class Wesen extends Thread {
    protected Welt    dieWelt;
    protected int     lpos; //  ``Laengengrad``
    protected int     bpos; //  ``Breitengrad``

    Wesen( Welt w, int l, int b ){
        dieWelt = w; lpos = l; bpos = b;
    }
    public void run() {
        do {
            try {
                sleep( schlafdauer() );
            } catch( InterruptedException ie ){ }
        } while( !istHandelnNoetig() );
        handeln();
    }
    protected abstract boolean istHandelnNoetig();
    protected abstract void handeln();

    private static int schlafdauer() { // in Millisekunden
        return (int)(Math.round(Math.random()*5000));
    }
}

class Lebewesen extends Wesen {
    Lebewesen( Welt w, int l, int b ){ super(w,l,b); }

    protected boolean istHandelnNoetig() {
        int i = dieWelt.anzahlNachbarn( lpos, bpos );
        return (i<2) || (3<i) ;
    }
    protected void handeln() {
        Wesen w = new Leerwesen(dieWelt,lpos,bpos);
        dieWelt.setWesen( lpos, bpos, w );
        w.start();
    }
}

class Leerwesen extends Wesen {
    Leerwesen( Welt w, int l, int b ){ super(w,l,b); }

    protected boolean istHandelnNoetig() {
        return dieWelt.anzahlNachbarn(lpos,bpos) == 3 ;
    }
    protected void handeln() {
        Wesen w = new Lebewesen(dieWelt,lpos,bpos);
        dieWelt.setWesen( lpos, bpos, w );
        w.start();
    }
}

```

Abbildung 6.3: Die Klassen der Wesen



```

class Welt {
    final static int MXG = 20; // maximale Gradzahl
    private Wesen[][] weltFeld;
    private Oberflaeche oberfl;

    Welt() {
        weltFeld = new Wesen[MXG][MXG];
        for( int l = 0; l<MXG; l++ ) {
            for( int b = 0; b<MXG; b++ ) {
                if( Math.random() < 0.25 ) {
                    weltFeld[l][b] = new Lebewesen(this,l,b);
                } else {
                    weltFeld[l][b] = new Leerwesen(this,l,b);
                }
            }
        }
        oberfl = new Oberflaeche( this );

        for( int l = 0; l<MXG; l++ ) {
            for( int b = 0; b<MXG; b++ ) {
                weltFeld[l][b].start();
            }
        }
        System.out.println("... und er sah, dass es gut war");
    }

    private int lebt( int l, int b ) {
        if( weltFeld[l%MXG][b%MXG] instanceof Lebewesen )
            return 1;
        return 0;
    }

    int anzahlNachbarn( int lp, int bp ) {
        int l = lp + MXG;
        int b = bp + MXG;

        return lebt(l-1,b-1) + lebt(l-1,b) + lebt(l-1,b+1)
            + lebt(l,b-1) + lebt(l,b+1)
            + lebt(l+1,b-1) + lebt(l+1,b) + lebt(l+1,b+1);
    }

    void setWesen( int lp, int bp, Wesen w ) {
        weltFeld[ lp ][ bp ] = w;
        oberfl.aktualisieren( lp, bp );
    }

    Wesen getWesen( int lp, int bp ) {
        return weltFeld[ lp ][ bp ];
    }
}

```

Abbildung 6.4: Die Welt-Klasse

**Kommunikation von Threads über den zentralen Speicher.** Der *zentrale Speicher* bei der Ausführung eines objektorientierten Programms besteht aus den Instanzvariablen der Objekte sowie aus den Klassenattributen. Da jeder Thread einen eigenen Laufzeitkeller besitzt, d.h. die Methodenparameter und lokalen Variablen der Methoden selbst verwaltet, rechnen wir Parameter und lokale Variablen nicht zum zentralen Speicher.

*zentraler  
Speicher*

Als *gemeinsamen Speicher* zweier Threads bezeichnen wir den Teil des zentralen Speichers, der von beiden Threads aus zugreifbar ist. Der gemeinsame Speicher zweier Threads besteht also aus den Instanzvariablen derjenigen Objekte, die von beiden Threads aus erreichbar sind, sowie aus den für die Threads sichtbaren Klassenattributen. Oft sagen wir auch, dass die Objekte, deren Instanzvariablen zum gemeinsamen Speicher gehören, im gemeinsamen Speicher liegen. Der gemeinsame Speicher macht einen Datenaustausch zwischen Threads sehr einfach: Die Threads können wechselseitig Daten schreiben und lesen.

*gemeinsamer  
Speicher*

Leider steht diesem recht einfachen Mechanismus ein technisches Problem im Wege. Werden Berechnungen real parallel ausgeführt, also auf unterschiedlichen Prozessoren zur Anwendung gebracht, so steht diesen je nach ausführender Hardware nicht unbedingt ein gemeinsamer Speicher zur Verfügung. Um dennoch unabhängig von der ausführenden Maschine jedem Thread Zugriff auf alle nötigen Variablen im gemeinsamen Speicher zu ermöglichen, definiert die Sprache Java ein eigenes *Speichermodell* (memory model), welches von der virtuellen Maschine passend auf der jeweiligen Hardware umgesetzt wird. Leider fehlt diesem Speichermodell eine Eigenschaft, die beim Programmieren gerne als implizit gegeben angenommen wird: die *sequentielle Konsistenz* (sequential consistency). Wir sprechen von sequentieller Konsistenz, wenn die Änderungen, die ein Thread an einer Variablen ausführt, unmittelbar auch bei allen anderen Threads in Erscheinung treten, die diese Variable lesen können.

Im folgenden Beispiel wird das Ende eines Threads AnhaltbarerThread über den Zustand einer Variable anhalten gesteuert.

```
public class Test {
    public static void main(String[] args) {
        AnhaltbarerThread thread = new AnhaltbarerThread();
        thread.start();
        thread.anhalten();
    }
}

class AnhaltbarerThread extends Thread {
    boolean anhalten = false;

    public void run() {
```

```

        while(!anhalten) {
        }
        System.out.println("Beendet");
    }

    public void anhalten() {
        anhalten = true;
    }
}

```

Wegen der fehlenden sequentielle Konsistenz ist nicht sichergestellt, dass `AnhaltbarerThread` tatsächlich beendet wird, denn das Speichermodell gibt keine Garantie, dass die Änderung der Instanzvariable `anhalten` durch den Thread der `main`-Methode tatsächlich auch im neu gestarteten Thread sichtbar wird.

*volatile*

Um dieses häufig gewünschte Verhalten dennoch zu erreichen, bietet die Sprache Java das `volatile`-Schlüsselwort, welches bei Deklarationen von Instanz- und Klassenvariablen mit angegeben werden kann. Für eine `volatile`-Variable stellt die virtuelle Maschine sicher, dass beim Lesen dieser Variable genau der Wert zurückgegeben wird, der (von welchem Thread auch immer) zuletzt der Variable zugewiesen wurde. Die Lösung für obiges Beispiel würde also darin bestehen, die Deklaration von `anhalten` folgendermaßen abzuändern:

```
volatile boolean anhalten = false;
```

Während Javas Speichermodell das Problem beschreiben kann, dass eine Zuweisung an eine Variable nicht in einem anderen Thread sichtbar wird, gibt es auch noch die umgekehrte Problematik, dass ein anderer Thread zu einem ungünstigen Zeitpunkt eine Zuweisung durchführt, obwohl es die Intention des Programmierers war, noch den alten Wert auszulesen. Diese Problematik und ihre Lösung – die Synchronisation – werden wir im Abschn. 6.2.2 behandeln.

**Kommunikation von Threads über Unterbrechungen.** Ein Thread kann einen anderen Thread *t* durch Aufruf der Methode `interrupt` der Klasse `Thread` unterbrechen (`t.interrupt()`; ) und ihn damit auf eine bestimmte Sache (z.B. das Vorhandensein bestimmter Daten im gemeinsamen Speicher) aufmerksam machen. Die Methoden der Klasse `Thread` zum Auslösen und Abfragen von Unterbrechungen sind:

```

void interrupt()
boolean isInterrupted()
static boolean interrupted()

```

Befindet sich der Thread  $t$ , auf dem `interrupt` aufgerufen wurde, weder im Zustand „schlafend“ noch im Zustand „wartend“, wird durch Setzen eines Flags vermerkt, dass für  $t$  eine Unterbrechungsanforderung besteht. Durch Aufruf von `isInterrupted` kann der Thread  $t$  feststellen, ob das Abbruchflag gesetzt wurde. Unterbleibt in  $t$  ein solcher Test, bleibt die Unterbrechungsanforderung ohne Wirkung.

Anders verhält es sich, wenn sich  $t$  im Zustand „schlafend“ oder im Zustand „wartend“ befindet. Dann wird  $t$  durch die Unterbrechung in den Zustand rechenbereit versetzt. Sobald er wieder ans Rechnen kommt, terminiert der aktuelle Aufruf von `sleep` bzw. `wait` in  $t$  abrupt mit einer `InterruptedException`<sup>7</sup>. Allerdings kann der Aufruf von `wait` nur weiter bearbeitet werden und terminieren, wenn  $t$  vorher den Monitor betreten konnte (siehe Abschn. 6.2.2).

Die statische Methode `interrupted` stellt den Status des Abbruchsflags beim *aktuell ausgeführten Thread* fest. Ihr Aufruf entspricht dem Aufruf von `Thread.currentThread().isInterrupted()`, setzt aber zusätzlich das Abbruchflag zurück.

Als Beispiel für die Benutzung des Unterbrechungsmechanismus erweitern wir unsere Realisierung vom Spiel des Lebens. Der Benutzer soll über die Oberfläche die Möglichkeit erhalten, in das Weltgeschehen eingreifen zu können. Dazu sei auf der Oberfläche jedem Feldelement eine Schaltfläche zugeordnet (vgl. Abb. 6.2). Das Anklicken der Schaltfläche soll neues Leben auf dem Feldelement erzeugen, wenn dieses leer ist, und andernfalls das Lebewesen durch ein Lebewesen ersetzen. Dazu genügt es, das aktuelle Wesen auf dem Feldelement zum Handeln zu bewegen. Dies lässt sich wohl am einfachsten durch Verwendung des Unterbrechungsmechanismus realisieren. Der Thread, der die `actionPerformed`-Methode der Schaltfläche ausführt, unterbricht den Thread, der das Wesen repräsentiert. Ein entsprechendes Fragment einer Schaltflächenklasse könnte wie folgt aussehen, wobei die Attribute `dieWelt`, `lpos` und `bpos` dieselbe Bedeutung haben wie in Klasse `Wesen`:

```
class WButton extends Button implements ActionListener {
    ...
    public void actionPerformed( ActionEvent e ) {
        dieWelt.getWesen(lpos,bpos).interrupt();
    }
}
```

<sup>7</sup>Achtung, beim Auslösen einer `InterruptedException` wird das Abbruchflag zurückgesetzt, sodass die Unterbrechung eigentlich direkt an der Stelle bearbeitet werden muss, an der die Ausnahme auftritt. Alternativ kann an dieser Stelle durch einen erneuten Aufruf von `interrupt` das Flag wieder gesetzt und dann später bearbeitet werden, oder die Ausnahme an die ggf. umfassende Methode weitergeleitet werden. Für weitere Details und Beispiele sei auf [Krü07] verwiesen.

Die von der Schaltfläche erzeugte Unterbrechung muss in der Klasse `Wesen` behandelt werden. Dazu führen wir eine boolesche Variable `unterbrochen` ein, die auf `true` gesetzt wird, wenn das Wesen unterbrochen wurde. Bei der turnusmäßigen Abfrage, ob Handeln nötig ist, muss dann auch eine mögliche Unterbrechung berücksichtigt werden:

```
abstract class Wesen extends Thread {
    ...
    public void run() {
        boolean unterbrochen = false;
        do {
            try {
                sleep( schlafdauer() );
            }
            catch( InterruptedException ie ){
                unterbrochen = true;
            }
        (+) if ( interrupted() )
                unterbrochen = true;
        } while( !istHandelnNoetig() && !unterbrochen );
        handeln();
    }
    ...
}
```

Das Beispiel zeigt beide Arten der Unterbrechungsbehandlung. Tritt eine Unterbrechung beim Schlafen auf, wird sie in der `catch`-Klausel behandelt. Tritt eine Unterbrechung an anderen Stellen während der Ausführung der Schleife auf, z.B. im Aufruf der Methode `schlafdauer`, wird sie in der mit (+) gekennzeichneten `if`-Anweisung erkannt.

Abbildung 6.5 zeigt eine entsprechende Implementierung für die „Welt“-Oberfläche und bietet die vollständige Klassendeklaration von `WButton`. Der Einfachheit halber wird die Welt als ein Feld von  $MXG \times MXG$  Schaltflächen dargestellt, die mittels eines `GridLayout-Manager`s angeordnet werden. Der Preis für diese einfache Implementierung besteht darin, dass das Programm bei einer größeren Anzahl von Feldelementen schnell recht ineffizient wird, da der Verwaltungsaufwand für eine Schaltfläche nicht unerheblich ist. Beim Aktualisieren eines Feldelements wird die Hintergrundfarbe der zugehörigen Schaltfläche je nachdem, was für ein Wesen dort lebt, auf Blau (Leerwesen) oder Gelb (Lebewesen) gesetzt.

```

class Oberflaeche extends BaseFrame {
    Welt dieWelt;
    Panel p;

    Oberflaeche( Welt w ) {
        dieWelt = w;
        p = new Panel();
        p.setLayout( new GridLayout( w.MXG, w.MXG ) );

        for( int l = 0; l < w.MXG; l++ ) {
            for( int b = 0; b < w.MXG; b++ ) {
                p.add( new WButton( w, l, b ) );
            }
        }
        add( p );
        setVisible( true );
    }
    void aktualisieren( int lp, int bp ){
        ((WButton) p.getComponent( lp*dieWelt.MXG+bp ))
            .aktualisieren();
        repaint();
    }
}

class WButton extends Button
    implements ActionListener {
    private Welt dieWelt;
    private int lpos;
    private int bpos;

    WButton( Welt w, int l, int b ) {
        dieWelt = w; lpos = l; bpos = b;
        setBackground( Color.blue );
        addActionListener( this );
    }

    public void actionPerformed((ActionEvent e) {
        dieWelt.getWesen( lpos, bpos ).interrupt();
    }

    public void aktualisieren() {
        Wesen w = dieWelt.getWesen( lpos, bpos );
        if( w instanceof Lebewesen ) {
            setBackground( Color.yellow );
        } else {
            setBackground( Color.blue );
        }
    }
}

```

Abbildung 6.5: Implementierung der „Welt“-Oberfläche

**Scheduling von Threads.** Die Sprachspezifikation von Java macht keine Aussagen darüber, wie die rechenbereiten Threads zu schedulen sind; d.h. dass Entwickler von Java-Übersetzern und -Laufzeitumgebungen diesbzgl. an keine Vorgaben gebunden sind. Insbesondere wird kein faires Scheduling verlangt. Es wird nicht einmal gefordert, dass einem rechnenden Thread nach einer bestimmten Zeit der Prozessor entzogen werden muss, wenn sich andere Threads im Zustand „rechenbereit“ befinden.

Diese großen Freiheitsgrade des Scheduling müssen bei der Entwicklung von Java-Programmen mit parallelen Threads sorgfältig berücksichtigt und ggf. durch geeignete Maßnahmen eingeschränkt werden (z.B.: Synchronisation; Freigabe des Prozessors mittels der Methode `yield`). Andernfalls erhält man leicht ein völlig unbeabsichtigtes Programmverhalten. Die sich daraus ergebenden Fehler fallen häufig erst auf, wenn man Programme, die bislang auf einem Rechensystem ohne erkennbare Probleme gelaufen sind, auf einem anderen Rechensystem ausführt, das ein anderes Scheduling-Verhalten hat. Vor allem im Zusammenhang mit der Internet-Programmierung, bei der man normalerweise die Rechensysteme nicht kennt, auf denen die Programme zur Ausführung kommen, kann der Nichtdeterminismus, der aus der Parallelität bzw. den Freiheitsgraden des Scheduling resultiert, erhebliche Probleme verursachen.

Zur Illustration des Fehlerpotentials, das sich durch unzureichende Einschränkung der Freiheitsgrade des Schedulers ergeben kann, betrachten wir unsere Realisierung vom Spiel des Lebens:

- Da der Scheduler frei unter den rechenbereiten Threads wählen kann, könnte er einige Wesen *verhungern* lassen (im Engl. spricht man von *starvation*), d.h. ihnen nie einen Prozessor zuteilen. Damit würde die Entwicklung in bestimmten Bereichen der Welt zum Stillstand kommen.
- Die Regeln zur Entwicklung der Population werden im Allgemeinen nicht eingehalten: Beispielsweise könnte ein Wesen mitten in der Ausführung der Methode `anzahlNachbarn` unterbrochen werden. Während es darauf wartet, weiterrechnen zu können, stirbt ein schon gezähltes Lebewesen, sodass die Methode am Ende ein falsches Ergebnis liefert.

*verhungern*

Je nach Scheduler läuft die Populationsentwicklung also regelmäßig ab oder bricht die Regeln. Dieses vage, fehlerträchtige Verhalten mag bei einem Spiel noch hinnehmbar sein, bei kommerzieller und vor allem sicherheitskritischer Software ist es jedoch völlig inakzeptabel. Der Programmentwickler hat dafür Sorge zu tragen, dass nur die Scheduling möglich sind, die zu einem korrekten Programmverhalten führen. Dazu stehen ihm im Wesentlichen Synchronisationsmechanismen zur Verfügung: Er kann die Threads bzgl. kritischer Ausführungsbereiche synchronisieren. Mechanismen zur Synchronisation stellt der folgende Abschnitt vor.

## 6.2.2 Synchronisation

Dieser Abschnitt besteht aus vier Teilen. Der erste Teil bietet eine knappe Einführung in die Problemquellen bzw. -klassen, die durch Synchronisation zu lösen sind. Der zweite Teil stellt die objektorientierte Variante des Monitorkonzepts vor, die in Java realisiert ist. Der dritte Teil demonstriert die Anwendung des Monitorkonzepts anhand von Beispielen und skizziert, wie man sich davon überzeugen kann, dass ein Programm bestimmte Synchronisationseigenschaften besitzt.

### 6.2.2.1 Synchronisation: Problemquellen

Synchronisation soll den Nichtdeterminismus nebenläufiger Threads so einschränken, dass alle verbleibenden Ausführungsmöglichkeiten zu einem korrekten Programmverhalten führen. Im Wesentlichen sind im Zusammenhang mit der Synchronisation vier mögliche Problemquellen relevant:

1. Gebrauch gemeinsamer Ressourcen, vor allem gemeinsamer Variablen.
2. Anforderungsgerechte Kooperation von Threads.
3. Gerechte Verteilung der Ausführung aller Threads: Fairness bzw. Verhungern.
4. Unbedachte Synchronisation: Verklemmung.

*Verklemmung*

Die ersten drei Problemquellen ergeben sich aus der zu lösenden Aufgabenstellung. Wir werden sie in den folgenden Absätzen etwas näher erläutern. Die vierte Problemquelle ergibt sich aus der Anwendung von Synchronisationstechniken: Eine unbedachte Einschränkung der Ausführungsmöglichkeiten von Threads kann dazu führen, dass alle Threads gegenseitig aufeinander warten und die weitere Ausführung dadurch blockiert ist. Auf dieses Problem gehen wir in Abschnitt 6.2.2.3 auf Seite 408 näher ein.

Verklemmungsfreiheit und Nicht-Auftreten von Verhungern ähneln sich auf den ersten Blick. Zur Unterscheidung ist es hilfreich zu untersuchen, ob es für ihre Spezifikation reicht, sich auf einen Zustand zu beziehen, oder ob man über die Existenz bestimmter Zustände in der Zukunft etwas aussagen muss. Verklemmungsfreiheit ist eine sogenannte *Sicherheitseigenschaft* eines Systems von Ausführungssträngen: In jedem Zustand soll gelten, dass einer der Stränge einen Fortschritt machen kann. Nicht-Auftreten von Verhungern ist eine sogenannte *Lebendigkeitseigenschaft*: Für jeden Strang soll es einen Zustand in der Zukunft geben, in dem er einen Fortschritt machen kann.

*Sicherheits- u.  
Lebendigkeitseigenschaft*

**Gebrauch gemeinsamer Ressourcen.** Synchronisation ist notwendig, um zu verhindern, dass Ausführungsstränge, die eine gemeinsame Ressource



nutzen wollen, sich gegenseitig stören. Bei der Thread-Programmierung in Java ist der lesende bzw. schreibende Zugriff auf den gemeinsamen Speicher die häufigste Form des Gebrauchs gemeinsamer Ressourcen. Ein typischer Konflikt sieht wie folgt aus: Ein Thread greift auf eine Instanzvariable zu, die von beiden Threads aus erreichbar ist, und schickt sich an, in Abhängigkeit vom Wert der Variablen eine entsprechende Operation auszuführen, wird aber vor der Ausführung der Operation vom Scheduler unterbrochen. Ein zweiter Thread verändert daraufhin den Wert der Variablen. Setzt der erste Thread dann seine Ausführung fort, wird er im Allgemeinen eine Operation ausführen, die nicht zum aktuellen Wert der Variablen passt.

Als Beispiel betrachten wir eine rudimentäre Kontenverwaltung in einer Bank (siehe Abb. 6.6). Die Konten sind der Einfachheit halber als ein Feld von ganzen Zahlen realisiert; d.h. jeder Feldkomponente entspricht ein Konto. Eine Methode `transfer` gestattet es, einen Betrag von einem Konto auf ein anderes umzubuchen. Sie prüft vorab, ob der Kontostand ausreicht.

```
class Bank {
    private int[] konten;
    ...
    boolean transfer( int von, int nach, int betrag ) {
        if( konten[von] >= betrag ) {
            konten[von] = konten[von] - betrag;
            konten[nach] = konten[nach] + betrag;
            return true;
        }
        return false;
    }
}
```

Abbildung 6.6: Ein Ausschnitt der Klasse `Bank`

Liegt ein `Bank`-Objekt im gemeinsamen Speicher zweier Threads *A* und *B*, könnte Folgendes passieren: *A* möchte zweitausend Euro von Konto 7 nach Konto 10 transferieren und startet die Ausführung der Methode `transfer`. Da der Kontostand von Konto 7 fünftausend Euro beträgt, ist die Bedingung der `if`-Anweisung erfüllt und der Ausdruck auf der rechten Seite der ersten Zuweisung wird ausgewertet und ergibt dreitausend. Bevor die Zuweisung ausgeführt wird, wird *A* vom Scheduler unterbrochen, und *B* kann seinen Transfer vornehmen, sagen wir dreitausend Euro von Konto 7 nach Konto 3. Wir nehmen an, dass *B* den Transfer ohne Unterbrechung abschließt und demnach Konto 7 einen Betrag von zweitausend Euro aufweist. Bei Fortführung des Threads *A* wird dem Konto 7 der von ihm vor der Unterbrechung berechnete Wert dreitausend zugewiesen. Dadurch ist der Inhaber von Konto 7 auf magische Weise um dreitausend Euro reicher geworden, denn das Konto hätte nach korrekter Ausführung beider Transfers einen Stand von

null Euro aufweisen müssen:  $(5000 - 2000) - 3000 = 0$ . Außerdem wurde eine erwartete Invariante der Methode `transfer` verletzt, da die Summe über alle Konten sich verändert hat.

Um besser nachvollziehen zu können, wie der fehlerhafte Ablauf im Detail aussieht, führen wir im Programmcode von `transfer` eine Hilfsvariable `nk` ein, die zunächst die neu berechneten Kontostände aufnimmt und die nach Berechnung des jeweils neuen Saldos dann dem entsprechenden Konto zugewiesen wird. (Auch wenn eine solche Hilfsvariable nicht explizit programmiert ist, wird sie temporär zur Auswertung der Ausdrücke angelegt.)

```
class Bank {
    private int[] konten;
    ...
    boolean transfer( int von, int nach, int betrag ) {
        if( konten[von] >= betrag ) {
            int nk = konten[von] - betrag;
            konten[von] = nk;
            nk = konten[nach] + betrag;
            konten[nach] = nk;
            return true;
        }
        return false;
    }
}
```

Hier der Ablauf noch einmal im Detail. Der Thread *A* beginnt die Ausführung der Methode `transfer`, wird aber nach der ersten Zuweisung

```
int nk = konten[von] - betrag;
```

unterbrochen. Nun beginnt der Thread *B* zu laufen, führt alle Anweisungen in der `transfer`-Methode aus und wird erst danach wieder unterbrochen. Schließlich führt *A* seine noch verbleibenden Anweisungen aus. Die folgende Tabelle zeigt, dass nach diesem Ablauf Konto 7 tatsächlich einen Saldo von dreitausend Euro statt null Euro aufweist. Die ersten beiden Spalten enthalten die von den beiden Threads ausgeführten Anweisungen in der oben beschriebenen Reihenfolge. In den letzten drei Spalten vermerken wir jeweils den Wert, der durch diese Anweisung verändert wird. Nicht aufgenommen wurden die Werte für die Konten 10 und 3, da uns bei dieser Demonstration nur der Wert von Konto 7 interessiert. Deren Kontostand sei bei Eintritt in die Methode `transfer` jeweils 1000 Euro. Um die Methodenparameter `von`, `nach` und `betrag` nicht auch noch mit in die Tabelle aufnehmen zu müssen, tragen wir deren Werte direkt in die angegebenen Anweisungen ein. Die Auswertung des Ausdrucks der `if`-Anweisung wurde der Einfachheit halber weggelassen. Er wird in beiden Aufrufen zu `true` ausgewertet.

Wie man sieht, enthält `konten[7]` nach Ausführung beider `transfer`-Methoden den Wert 3000.

Thread A	Thread B	konten[7]	nk in transfer von A	nk in transfer von B
		5000		
int nk = konten[7]-2000;			3000	
	int nk = konten[7]-3000			2000
	konten[7]= nk;	2000		
	nk = konten[3]+3000;			4000
	konten[3] = nk;			
	return true;			
konten[7]= nk;		3000		
nk = konten[10]+2000;			3000	
konten[10] = nk;				
return true;				

Abbildung 6.7: Falscher Kontostand aufgrund fehlender Synchronisation

Das Beispiel demonstriert die Notwendigkeit zu verhindern, dass bestimmte Aktionsfolgen unterschiedlicher Threads überlappt ausgeführt werden. Derartige unteilbare Aktionsfolgen nennt man häufig auch *kritische Bereiche*. Ein weiteres Beispiel für kritische Bereiche haben wir im Zusammenhang mit der Methode `anzahlNachbarn` beschrieben (vgl. die Erläuterungen am Ende von Abschn. 6.2.1.2).

*kritischer  
Bereich*

**Kooperation.** Synchronisation ist außerdem gefordert, wenn zwei oder mehrere Threads miteinander kooperieren wollen. Dann ist es häufig notwendig, dass ein Thread wartet, bis ein anderer eine bestimmte Leistung erbracht hat. Ein typisches Beispiel für kooperierende Threads liefert das Erzeuger-Verbraucher-Problem: Ein Thread erzeugt Gegenstände und legt sie in einem Puffer ab. Er muss warten, wenn der Puffer voll ist. Ein anderer Thread holt sich die Gegenstände aus dem Puffer und verbraucht sie. Dieser muss warten, wenn der Puffer leer ist. Der Wartezustand des jeweils anderen Threads kann nur vom kooperierenden Thread beendet werden.

**Fairness und Verhungern.** Die Gewährleistung von Fairness ist eigentlich eine Aufgabe des Schedulers. Da die Sprachspezifikation von Java aber nicht fordert, dass der implementierte Scheduling-Mechanismus fair sein muss, muss der Programmierer diese Aufgabe so weit wie möglich übernehmen. Er kann dazu zum Beispiel in allen Threads an geeigneten Stellen für den Aufruf der Methode `yield` sorgen. `yield` signalisiert dem Scheduler, dass der laufende Thread den Prozessor freigeben will.

Verhungern kann statt durch fehlende Fairness auch durch eine fehlerhafte Synchronisation entstehen. Als einfaches Beispiel dazu betrachten wir drei

Threads, die den Zugang zu einem kritischen Bereich mit Hilfe eines Tokens koordinieren: Nur der Thread, der den Token hat, darf den kritischen Bereich betreten. Beim Verlassen des kritischen Bereichs gibt ein Thread den Token an einen anderen Thread weiter. Wechseln sich zwei Threads im Besitz des Tokens ab, verhungert der dritte, auch wenn die Scheduling-Strategie fair ist.

### 6.2.2.2 Ein objektorientiertes Monitorkonzept

Für die Lösung von Synchronisationsaufgaben stellt Java Konstrukte zur Realisierung von Monitoren zur Verfügung. Unter einem *Monitor* stellt man sich am besten eine Überwachungseinheit vor, die den Zugang zu bestimmten Anweisungsfolgen kontrolliert. In Java kann man Blöcke und Methoden von einem Monitor überwachen lassen. Um bei der Erläuterung nicht immer zwischen diesen beiden Fällen unterscheiden zu müssen, betrachten wir zunächst nur den Mechanismus zur Überwachung der Ausführung von Methoden. Die Synchronisation von Blöcken behandeln wir dann im Anschluss. Zur Abkürzung der Sprechweise bezeichnet man häufig einen Methodenauf-ruf, der vom Monitor zugelassen wird, als *Betreten des Monitors*.

*Monitor*

*Betreten  
eines Monitors*

Monitore bilden ein Synchronisationskonzept, das Anfang der siebziger Jahre von C. A. R. Hoare für prozedurale Sprachen ausgearbeitet wurde (siehe [Hoa74] bzw. [BA90]). Im Folgenden beschreiben wir die objektorientierte Variante des Monitorkonzepts, die Java zugrunde liegt. Dazu erläutern wir zunächst das Konzept zur Sperrung von Monitoren und beschreiben dann den erweiterten Mechanismus, der durch die Methoden `wait` und `notify` realisiert ist. Auf die Anwendung des Monitorkonzeptes zur Synchronisierung von Threads gehen wir in Unterabschn. 6.2.2.3 ein.

**Sperren von Monitoren.** Das Monitorkonzept für objektorientierte Sprachen sieht vor, dass jedes Objekt einen Monitor besitzt. Dieser Monitor überwacht die Ausführung derjenigen Methoden des Objekts, die als synchronisiert deklariert wurden. Eine Methode heißt *synchronisiert*, wenn ihrer Deklaration das Schlüsselwort `synchronized` vorangestellt ist. Kernbestandteil des objektorientierten Monitorkonzepts ist ein Mechanismus, der es Threads ermöglicht, Monitore mit einer Sperre zu versehen:

*synchronisiert*

*Hat Thread A z.B. durch Aufruf einer synchronisierten Methode auf einem Objekt X den Monitor von X betreten, können andere Threads solange keine synchronisierten Methoden auf X aufrufen, bis Thread A den Monitor wieder verlassen hat.*

Java unterstützt auch die Synchronisation von Klassenmethoden. Den Monitor für Klassenmethoden besitzt das sogenannte *Class-Objekt* der Klasse, in der die Klassenmethode deklariert wurde. Ein Class-Objekt beschreibt die zugehörige Klasse. Man kann es fragen, welche Konstruktoren die Klasse besitzt, welche Methoden sie bereitstellt, welche Schnittstellen sie implemen-

*Class-  
Objekt*

tiert etc. Für mehr Details und den Einsatzzweck solcher Objekte sei auf [PH00] verwiesen und auf [Krü07], Kapitel 43.

Eine falsche Verwendung von Monitoren kann leicht zu schwerwiegenden und schwer auffindbaren Fehlern führen. Deshalb ist ein präzises Verständnis ihrer Funktionsweise Voraussetzung für den Einsatz dieses relativ komplexen Sprachkonstrukts. Grundsätzlich gilt Folgendes:

- Der Monitor eines Objektes kann maximal von einem Thread gesperrt werden, von diesem allerdings mehrfach. Der Monitor des Objekts merkt sich, von welchem Thread und wie oft er gesperrt wurde.
- Ein Thread kann die Monitore mehrerer Objekte sperren. Er merkt sich, welche Monitore und wie oft er diese Monitore gesperrt hat.

Das relevante Szenario zum Verständnis des Monitormechanismus besteht darin, dass ein Thread  $A$  auf einem Objekt  $X$  eine synchronisierte Methode  $m$  aufrufen möchte<sup>8</sup>. Der Monitor von  $X$  prüft, ob er bereits von einem anderen Thread betreten wurde. In diesem Fall wird der Thread  $A$  in den Zustand „Monitor-blockiert“ versetzt (vgl. Abb. 6.1, S. 385). Andernfalls wird der Monitor des Objektes  $X$  von  $A$  gesperrt, wenn er vorher frei war. War er vorher schon von  $A$  gesperrt, wird die Anzahl der Sperrungen durch  $A$  um eins erhöht. Danach kann  $A$  den Monitor von  $X$  betreten, d.h. der Monitor gibt die Ausführung von  $m$  für  $A$  frei. Nachdem  $A$  den Rumpf von  $m$  ausgeführt hat, wird die Sperrung des Monitors von  $X$  wieder aufgehoben bzw. die Anzahl der Sperrungen um eins erniedrigt. Wurde die Sperrung aufgehoben, wird außerdem einer der Threads, der wegen der Sperrung des Monitors blockiert ist, wieder in den Zustand „rechenbereit“ gesetzt.

Um zu verstehen, warum man es einem Thread gestattet, einen Monitor mehrfach zu sperren, betrachten wir folgende Klassenskizze:

```
class K {
    ...
    synchronized void m() {
        ...
        n();
        ...
    }
    synchronized void n() { ... }
}
```

Ruft ein Thread  $A$  die Methode  $m$  auf einem  $K$ -Objekt  $X$  auf, dessen Monitor nicht gesperrt ist, sperrt er den Monitor von  $X$ . Zur Ausführung von  $m$  muss  $A$  auch die Methode  $n$  auf  $X$  ausführen. Da  $n$  auch als `synchronized` deklariert ist, muss  $A$  den Monitor von  $X$  wieder sperren. Da der aber bereits bei

<sup>8</sup>Für statische Methoden gilt Entsprechendes.

Eintritt in `m` von `A` gesperrt wurde, ist der Monitor nicht mehr frei. `A` müsste daher mitten in der Ausführung von `m` in den Zustand Monitor-blockiert für `X` versetzt werden. Da `A` den Monitor von `X` gesperrt hat, kann auch nur `A` diesen Monitor wieder frei geben. Das kann er aber nur, wenn er die Methode `m` beenden könnte, was nicht möglich ist, da er in der Ausführung vom `m` darauf warten muss, `n` ausführen zu dürfen. Dürfte ein Thread einen Monitor nicht mehrfach sperren, würde der Aufruf von `n` also zur dauernden Blockierung von `A` und damit zur dauernden Sperrung des Monitors von `X` führen.

Ein wichtiger Grund, das mehrfache Sperren von Monitoren zu erlauben, besteht also darin, es zu ermöglichen, dass synchronisierte Methoden andere synchronisierte Methoden des gleichen Objekts benutzen können.

**Synchronisierte Blöcke.** Neben synchronisierten Methoden unterstützt Java sogenannte synchronisierte Blöcke. Die Syntax dafür ist

```
synchronized ( Ausdruck ) Block
```

wobei der Ausdruck von einem Referenztyp sein muss. Ein synchronisierter Block wird von einem Thread `A` wie folgt ausgeführt: Zunächst wertet `A` den Ausdruck aus; das Ergebnisobjekt sei mit `X` bezeichnet. Der Monitor von `X` prüft dann, ob er bereits von einem anderen Thread gesperrt wurde, und verfährt danach ganz analog zu der Ausführung synchronisierter Methoden. Zur Illustration der Semantik synchronisierter Blöcke und des Zusammenhangs zu synchronisierten Methoden zeigen wir, wie man mit synchronisierten Blöcken Methoden synchronisieren kann. In der folgenden Methodendeklaration ist nicht die Methode selbst, sondern ihr Rumpf synchronisiert:

```
void mm() {
    synchronized( this ) {
        ... // Rumpf von mm
    }
}
```

Diese Methodendeklaration ist gleichbedeutend mit der folgenden:

```
synchronized void mm() {
    ... // Rumpf von mm
}
```

Synchronisierte Blöcke stellen das allgemeinere Sprachkonstrukt dar. Zum einen erlauben sie es, statt einer kompletten Methode nur einen kleineren Block von Anweisungen innerhalb einer Methode zu synchronisieren und somit die Sperrung des Monitors zu minimieren. Zum anderen erlauben sie es, die Ausführung eines Blocks in einer Klasse auch bzgl. Objekten anderer Klassen zu synchronisieren wie z.B. in der Methode `einzahlen` der Klasse `Bank` auf Seite 408. Hier wird zusätzlich zum Monitor des `this`-Objekts der Monitor von `konten` gesperrt.

**Synchronized und Volatile** Neben der Eigenschaft, dass ein durch einen bestimmten Monitor synchronisierter Codeabschnitt nur von einem Thread gleichzeitig durchlaufen werden kann, gilt eine weitere Zusage für durch Monitore kontrollierte Blöcke. Innerhalb synchronisierten Codes sichert Javas Speichermodell sequentielle Konsistenz zu, also bei Zugriffen auf Klassen- und Instanzvariablen stets alle inzwischen erfolgten Zuweisungen an diese von anderen Threads zu berücksichtigen. Für Klassen- und Instanzvariablen, auf welche nur synchronisiert zugegriffen wird, erübrigt sich somit eine Deklaration also `volatile`. Da bei lesendem und schreibendem Zugriff mehrerer Threads auf gemeinsamen Speicher in vielen Fällen ohnehin Synchronisation nötig ist, wird die Anwendungsmöglichkeit von `volatile` erheblich eingeschränkt. Dies und das recht unintuitive Fehlen der sequentiellen Konsistenz in Java werden wohl die Gründe dafür sein, dass selbst gestandene Java-Programmierer mitunter mit der korrekten Verwendung von `volatile` nicht vertraut sind.

**Der wait/notify-Mechanismus.** Für die Synchronisation von kooperierenden Threads erweist sich der beschriebene Sperrungsmechanismus als unzureichend. Wie beim oben skizzierten Erzeuger-Verbraucher-Problem kommt es bei kooperierenden Threads häufig vor, dass ein Thread eine bestimmte Bedingung prüfen muss und nur fortsetzen kann, wenn die Bedingung erfüllt ist. Andernfalls muss er warten, bis die Bedingung erfüllt ist. Da die Bedingung typischerweise von gemeinsamen Variablen abhängt, muss der Programmtext, der die Bedingung prüft, in vielen Fällen innerhalb einer synchronisierten Methode liegen.

Um zu verdeutlichen, warum der oben beschriebene Sperrungsmechanismus unzureichend ist, betrachten wir folgendes Szenario: Ein Thread *A* hat den Monitor eines Objektes *X* durch Aufruf einer synchronisierten Methode gesperrt und kommt nun an die Programmstelle innerhalb der synchronisierten Methode, an der die Bedingung zu prüfen ist. Die Bedingung ist nicht erfüllt, *A* muss also warten. Damit andere Threads den Monitor von *X* betreten können – dies ist häufig die Voraussetzung, um zu gewährleisten, dass die Bedingung zu einem späteren Zeitpunkt erfüllt ist –, muss *A* die Sperre des Monitors lösen, d.h. den Methodenaufruf verlassen, der zu dessen Sperrung geführt hat. Nach dem Verlassen des Methodenaufrufs wird er eine Weile warten, um dann die Methode erneut aufzurufen, den Monitor von *X* wieder zu betreten und zu prüfen, ob die Bedingung erfüllt ist. Dieser Vorgang des aktiven Wartens (engl. *busy waiting*) kann sich viele Male wiederholen und verschwendet unnötig Ressourcen. Mit dem wait/notify-Mechanismus ist es möglich, den gesperrten Objektmonitor freizugeben und *A* in einen Wartezustand zu versetzen, ohne dass *A* den begonnenen Methodenaufruf abbrechen muss. Ein anderer Thread kann dann *A* davon benachrichtigen, dass die Bedingung erfüllt ist.

Ein wait/notify-Mechanismus ist üblicherweise Bestandteil von Monitorrealisierungen, wobei die notify-Operation oft auch als „signal“ bezeichnet wird. Allerdings weichen die Realisierungen zum Teil nicht unerheblich voneinander ab. Beispielsweise gibt es Ausprägungen, bei denen die wait-Operation an eine bestimmte Bedingung gekoppelt wird, deren Erfüllung nach Beendigung des Wartens automatisch sichergestellt ist. Wie wir sehen werden, ist in Java eine wait-Operation an keine Bedingung gekoppelt, sondern nur eine Möglichkeit, einen Thread, der sich innerhalb einer synchronisierten Methode befindet, in einen Wartezustand zu versetzen. (Wie man trotzdem einen Zusammenhang zwischen dem wait-Aufruf und einer Bedingung herstellen kann, werden wir in Unterabschn. 6.2.2.3 erläutern.)

Beim wait/notify-Mechanismus von Java ist jedem Objekt eine Wartemenge zugeordnet. Mittels der Methode `wait` kann sich ein Thread in eine solche Wartemenge eintragen. Ein anderer Thread kann Threads aus der Wartemenge mittels der Methoden `notify` bzw. `notifyAll` wieder rechenbereit setzen.

Im Einzelnen funktioniert der Mechanismus wie folgt: Ein Thread *A* führt eine synchronisierte Methode eines Objekts *X* aus, *A* hat also den Monitor von *X* betreten. Nun kann sich *A* mittels eines Aufrufs `X.wait()` in die Wartemenge zu *X* eintragen, d.h. *A* versetzt sich in einen Wartezustand (Zustand „wartend“ in Abb. 6.1, S. 385). Dabei gibt *A* den gesperrten Objektmonitor von *X* frei und merkt sich, wie oft er diesen gesperrt hatte. Die anderen von *A* gesperrten Monitore werden nicht freigegeben.

Durch die Freigabe des Monitors erhalten andere Threads Zugang zu den synchronisierten Methoden der Klasse von *X* (bzw. zu Blöcken, die von *X* überwacht werden). Insbesondere kann ein anderer Thread *B* den Monitor von *X* betreten und Threads aus der Wartemenge von *X* benachrichtigen. Dafür stehen zwei Methoden zur Verfügung:

- Der Aufruf `X.notify()` wählt einen beliebigen<sup>9</sup> Thread aus der Wartemenge von *X* aus, wenn diese nicht leer ist, und setzt ihn wieder rechenbereit,
- der Aufruf `X.notifyAll()` setzt alle Threads aus der Wartemenge rechenbereit.

Bevor ein Thread *A*, der auf diese Weise wieder rechenbereit wurde, die Ausführung der wait-Methode beenden kann, die ihn in den Wartezustand versetzt hat, muss er den Monitor von *X* wieder betreten (und zwar mit der gleichen Anzahl von Sperrungen). Dabei konkurriert er in üblicher Weise mit

---

<sup>9</sup>Oft werden die Wartemengen als Warteschlangen realisiert; ausgewählt wird dann der zuerst eingetragene Thread. Die Java-Sprachbeschreibung erlaubt aber auch andere Realisierungen, so dass sich der Programmierer nicht auf eine Realisierung als Schlange verlassen kann.



anderen Threads. Gelingt ihm das Sperren, kann er den wait-Aufruf beenden und die Ausführung der synchronisierten Methode fortsetzen, in der der wait-Aufruf steht; andernfalls wird er in den Zustand „Monitor-blockiert“ versetzt.

Der wait-/notify-Mechanismus bietet für viele Synchronisationsaufgaben bei kooperierenden Threads eine flexible Unterstützung. Er birgt allerdings auch die Gefahr in sich, dass mehrere Threads gegenseitig aufeinander warten und es dadurch zu Verklemmungen kommt.

### 6.2.2.3 Synchronisation mit Monitoren

Dieser Unterabschnitt diskutiert anhand von zwei Beispielen die Anwendung des Monitor-Mechanismus zur Lösung von Synchronisationsaufgaben. Dabei werden wir auch jeweils informell grundlegende Korrektheitsaspekte betrachten. Dies machen wir zum einen, um das Verständnis des Monitor-konzepts und der Synchronisationsproblematik zu vertiefen, zum anderen, um die wichtigsten Beweismethoden wenigstens einmal am Beispiel darzustellen. Für eine detaillierte bzw. formale Behandlung der Verifikation von Korrektheitsaussagen müssen wir auf andere Literatur verweisen (siehe z.B. [BA90]).

**Wechselseitiger Ausschluss.** Konflikte beim Zugriff auf gemeinsame Variablen, wie wir sie anhand des Bankenbeispiels von Abb. 6.6 erläutert haben, lassen sich vermeiden, indem man Programmbereiche, die auf gemeinsame Variable zugreifen, synchronisiert. Betrachten wir dazu die vollständigere Version der Klasse `Bank` von Abb. 6.8. Wenn mehrere Threads Zugriff auf das gleiche `Bank`-Objekt haben, sind alle Feldelemente des Attributs `konten` gemeinsame Variablen dieser Threads. Es ist deshalb wichtig, wie im Beispiel gezeigt, *alle* Zugriffsoperationen auf diese gemeinsamen Variablen zu synchronisieren. Wären andernfalls beispielsweise nur `abheben` und `transfer` synchronisierte Methoden, `einzahlen` aber nicht, könnte Folgendes passieren: Ein Thread *A* ruft die Methode `abheben` für `Bank X` auf und wird nach der Auswertung von `konten[ktonr]-betrag` zu (z.B. zweitausend Euro) unterbrochen. Daraufhin ruft ein Thread *B* die Methode `einzahlen` auf. Da wir angenommen hatten, dass diese Methode nicht synchronisiert ist und also nicht vom Monitor überwacht wird, kann die Einzahlung vorgenommen werden, obwohl der Monitor gesperrt ist. Die Fortsetzung von *A* führt schließlich dazu, dass die Einzahlung verloren geht. *A* weist dann nämlich `konten[ktonr]` einfach wieder den von ihm vor der Unterbrechung ermittelten Betrag (von z.B. zweitausend Euro) zu. Man kann sich den Ablauf ähnlich überlegen wie in Abbildung 6.7.

Um die korrekte Ausführung kritischer Bereiche, d.h. unteilbarer Aktionsfolgen, zu gewährleisten, ist es im Allgemeinen notwendig, dass der

```

class Bank {
    private int[] konten;

    Bank() { ... }
    Bank( Bank zentrale ) { ... }

    synchronized void einzahlen( int ktonr, int betrag ) {
        konten[ktonr] = konten[ktonr]+betrag;
    }

    synchronized boolean abheben( int ktonr, int betrag ) {
        if( konten[ktonr] >= betrag ) {
            konten[ktonr] = konten[ktonr]-betrag;
            return true;
        }
        return false;
    }

    synchronized boolean transfer( int von, int nach, int b ){
        ...
    }
}

```

Abbildung 6.8: Die Klasse Bank

ausführende Thread zunächst *die Monitore aller* Objekte sperrt, auf deren Instanzvariablen im kritischen Bereich zugegriffen wird, und erst dann die Aktionen ausführt. Wie obiges Beispiel zeigt, muss darüber hinaus sichergestellt sein, dass *jeder* Zugriff auf diese Instanzvariablen innerhalb entsprechend synchronisierter Anweisungen erfolgt. Die genaue Realisierung dieser Regel ist im Allgemeinen nicht einfach und oft aus Effizienzgründen nur in abgeschwächter Form wünschenswert.

Um einige problematische Aspekte in diesem Zusammenhang zu diskutieren, betrachten wir nochmals die obige Klasse Bank. Sie verletzt die Regel insofern, als dass der Monitor des Feldobjektes `konten` von den Methoden nicht gesperrt wird. Solange kein anderes Bank-Objekt eine Referenz auf die Konten erhalten kann, schützt der Monitor des Bank-Objekts gleichzeitig das Konten-Objekt, so dass eine zusätzliche Sperrung des Monitors des Konten-Objekts nicht nötig ist. Anders verhält es sich, wenn mehrere Bankfilialen auf eine zentrale Kontenverwaltung zugreifen, wie im folgenden Programmfragment angedeutet:

```

class Bank {
    private int[] konten;

    Bank() { konten = new int[10]; }
    Bank( Bank zentrale ) { konten = zentrale.konten; }
    ...
}

```

```

    }

    public class BankTest {
        static Bank b1 = new Bank();
        static Bank b2 = new Bank( b1 );
        ...
    }

```

In diesem Fall werden Zugriffe auf die Konten über die Bank `b1` nicht mit den Zugriffen über die Bank `b2` synchronisiert, so dass es wieder zu den in Abschn. 6.2.2.1 geschilderten Inkonsistenzen kommen kann. Der obigen Regel folgend kann man diese Inkonsistenzen vermeiden, indem man zusätzlich in allen Methoden der Bank auch den Monitor des Feldobjektes `konten` sperrt. Wir demonstrieren dieses Vorgehen am Beispiel der Methode `einzahlen`:

```

    synchronized void einzahlen( int ktonr, int betrag ) {
        synchronized( konten ) {
            konten[ktonr] = konten[ktonr]+betrag;
        }
    }

```

Threads, die mehrere Monitore sperren müssen, um ihre kritischen Bereiche zu schützen, bergen allerdings eine relativ große Verklemmungsgefahr. Seien beispielsweise *th1* und *th2* Threads, die die Monitore der Objekte *x* und *y* sperren müssen. Nehmen wir an, dass *th1* zunächst den Monitor von *x* und dann den von *y* sperren will, während *th2* erst den Monitor von *y* und dann den von *x* zu sperren beabsichtigt. Dies ist z.B. der Fall in folgendem Beispiel:

```

class Thread1 extends Thread {
    Object x;
    Object y;

    Thread1 (Object x, Object y) {
        this.x = x;
        this.y = y;
    }

    public void run () {
        // Anweisungsblock 1
        synchronized (x) {
            // Anweisungsblock 2
            synchronized (y) {
                // Anweisungsblock 3
            }
        }
    }
}

```

```
}

class Thread2 extends Thread {
    Object x;
    Object y;

    Thread2 (Object x, Object y) {
        this.x = x;
        this.y = y;
    }

    public void run () {
        // Anweisungsblock 1
        synchronized (y) {
            // Anweisungsblock 2
            synchronized (x) {
                // Anweisungsblock 3
            }
        }
    }
}

class Verklemmung {
    public static void main(String[] args) {
        Object x = new Object();
        Object y = new Object();

        Thread th1 = new Thread1 (x, y);
        Thread th2 = new Thread2 (x, y);

        th1.start();
        th2.start();
    }
}
```

Wird einer der Threads nach der ersten Sperrung unterbrochen, sperrt der andere Thread den jeweils anderen Monitor, sodass beide Monitore gesperrt sind und weder *th1* noch *th2* fortsetzen kann. Wird beispielsweise *th1* im Anweisungsblock 2 unterbrochen, hat er den Monitor von *x* gesperrt, aber noch nicht den von *y*. Führt jetzt *th2* seine Anweisungsblöcke 1 und 2 aus, hat er bereits den Monitor von *y* gesperrt. Um den Anweisungsblock 3 ausführen zu können, muss *th2* den Monitor von *x* betreten, der aber von *th1* gesperrt ist. *th2* wechselt daher in den Zustand Monitor-blockiert. Jetzt wird *th1* wieder rechenbereit gesetzt und beendet seinen Anweisungsblock 2. Zur Ausführung des Anweisungsblocks 3 muss *th1* zunächst den Monitor von *y*

betreten. Da dieser aber von *th2* gesperrt ist, kann auch *th1* seine Ausführung nicht fortsetzen. Beide Threads warten also jetzt auf die Freigabe des Monitors, der durch den jeweils anderen Thread gesperrt wurde. Eine solche Situation kann man in obigen Beispiel dadurch herbeiführen, dass man z.B. in der *run*-Methode der Klasse *Thread1* direkt vor dem Anweisungsblock 2 eine *sleep*-Anweisung einfügt:

```
public void run () {
    // Anweisungsblock 1
    synchronized (x) {
        try {
            Thread.sleep(100);
        } catch (InterruptedException ex) {...}

        // Anweisungsblock 2
        synchronized (y) {
            // Anweisungsblock 3
        }
    }
}
```

Zur Vermeidung derartiger Verklemmungen sollte man die zu sperrenden Monitore ordnen und Sperrungen nur in der Reihenfolge der Ordnung vornehmen. Hätte man im Beispiel etwa festgelegt, dass der Monitor von *x* immer vor dem von *y* zu betreten ist, hätte Thread *th2* anders implementiert werden müssen und die geschilderte Verklemmungsmöglichkeit wäre dadurch verhindert worden.

**Kooperierende Threads: Erzeuger/Verbraucher.** Die Kommunikation zwischen Erzeugern und Verbrauchern über ein Lager beschränkter Größe ist ein typisches Beispiel für kooperierende Threads und hervorragend geeignet, die Anwendung des *wait/notify*-Mechanismus zu demonstrieren. Wir betrachten zunächst den Fall, dass es genau einen Erzeuger und genau einen Verbraucher gibt. Der Erzeuger ist als ein Thread realisiert, der Produkte erzeugt und in einem Ringpuffer ablegt:

```
class Erzeuger extends Thread {
    private RingPuffer lager;

    Erzeuger( RingPuffer rp ){ lager = rp; }

    public void run(){
        while( true ){
            Produkt prd = new Produkt();
            try{
                sleep( prd.getProduktionsZeit() );
            }
        }
    }
}
```

```

        lager.ablegen( prd );
    } catch( InterruptedException e ){
        System.out.println("Unzulaessige Unterbrechung!");
        System.exit( 0 );
    }
}
}
}
}

```

Die Synchronisation mit dem Verbraucher geschieht über den Puffer und seine Methoden. Aus Erzeugersicht ist das die Methode `ablegen`. Sie wartet, wenn der Puffer voll ist:

```

public synchronized void ablegen( Produkt prd )
                                throws InterruptedException {
    if( istVoll() ) wait();
    abspeichern( prd );
    notify();
}

```

Die Abbildungen 6.9 und 6.10 bieten die Realisierung des Verbrauchers, des Puffers und eines kleinen Programmrahmens.

Die korrekte Arbeitsweise des Programms hängt davon ab, dass

1. der Zugriff auf den gemeinsamen Speicher synchronisiert ist;
2. kein Produkt in einem vollen Puffer abgelegt wird;
3. nicht aus dem leeren Puffer gelesen wird;
4. keine Verklemmung von Erzeuger und Verbraucher auftreten kann.

Der erste Punkt bezieht sich auf die gemeinsamen Variablen zur Realisierung des Puffers. Da auf sie nur über die Methoden der Klasse `RingPuffer` zugegriffen werden kann und alle diese Methoden synchronisiert sind (vgl. die Diskussion von S. 408), ist die konsistente Behandlung sichergestellt, sofern die Anforderungen 2 und 3 erfüllt sind. Für die Anforderung 2 müssen wir garantieren, dass die Methode `abspeichern` nur aufgerufen wird, wenn der Puffer nicht voll ist. Betrachten wir dazu die zwei möglichen Szenarien:

1. Der Erzeuger betritt den Monitor des Ringpuffers, um ein Produkt abzulegen, und der Puffer ist nicht voll. Da zwischen der Abfrage und dem Abspeichern kein anderer Thread Zugriff auf den Ringpuffer hat und ihn anfüllen könnte, kann das Produkt korrekt abgespeichert werden.

```

import java.util.Random;

class Produkt {
    private static Random rg = new Random();
    private int produktionsZeit;
    Produkt() {
        int pz = rg.nextInt() % 1000;
        produktionsZeit = pz < 0 ? -pz : pz ;
    }
    int getProduktionsZeit() { return produktionsZeit; }
}

class Erzeuger extends Thread { ... // wie im Text }

class Verbraucher extends Thread {
    private RingPuffer lager;

    Verbraucher( RingPuffer rp ){ lager = rp; }

    public void run(){
        while( true ){
            try {
                Produkt prd = lager.holen();
                sleep( 500 ); // "Verbrauchszeit"
            } catch( InterruptedException e ){
                System.out.println("Unzulaessige Unterbrechung!");
                System.exit( 0 );
            }
        }
    }
}

public class ErzeugerVerbraucherTest {
    public static void main( String[] args ) {
        RingPuffer rpuf = new RingPuffer( 4 );
        Erzeuger e = new Erzeuger( rpuf );
        Verbraucher v = new Verbraucher( rpuf );
        e.start();
        System.out.println("Erzeuger nimmt Arbeit auf");
        v.start();
        System.out.println("Verbraucher nimmt Arbeit auf");
    }
}

```

Abbildung 6.9: Erzeuger und Verbraucher

```
class RingPuffer {
    private  Produkt[] puffer;
    private  int eingabeIndex, ausgabeIndex;
    private  int pufferGroesse;
    private  int gepuffert;      //  Anzahl gepufferter Elemente

    public RingPuffer( int groesse ){
        eingabeIndex = 0;
        ausgabeIndex = 0;
        gepuffert = 0;
        pufferGroesse = groesse;
        puffer = new Produkt[pufferGroesse];
    }

    public synchronized void ablegen( Produkt prd )
        throws InterruptedException {
        if( istVoll() ) wait();
        abspeichern( prd );
        notify();
    }

    public synchronized Produkt holen()
        throws InterruptedException {
        if( istLeer() ) wait();
        notify();
        return auslesen();
    }

    private synchronized boolean istVoll() {
        return gepuffert == pufferGroesse;
    }
    private synchronized boolean istLeer() {
        return gepuffert == 0;
    }
    private synchronized void abspeichern( Produkt prd ) {
        puffer[ eingabeIndex ] = prd;
        gepuffert++;
        eingabeIndex = (eingabeIndex+1) % pufferGroesse;
    }
    private synchronized Produkt auslesen() {
        Produkt prd = puffer[ ausgabeIndex ];
        gepuffert--;
        ausgabeIndex = (ausgabeIndex+1) % pufferGroesse;
        return prd;
    }
}
```

Abbildung 6.10: Der Ringpuffer



2. Der Erzeuger betritt den Monitor, um ein Produkt abzulegen, und der Puffer ist voll. Der Erzeuger geht also in den Wartezustand. Diesen kann er nur verlassen, wenn
  - (a) der Verbraucher `notify` aufgerufen oder
  - (b) eine Unterbrechung den Wartezustand beendet hat.

Fall (b) wird im diskutierten Programm als unzulässig betrachtet und führt zum Programmabbruch.

Fall (a) bedeutet, dass der Verbraucher die Methode `holen` ausgeführt hat. Da dies und jede weitere Ausführung von `holen` einen nicht vollen Puffer garantiert und kein anderer Zugriff auf das Lager erfolgen kann, da nur ein Erzeuger und ein Verbraucher existieren, ist Anforderung 2 erfüllt.

Die Argumentation bzgl. der Anforderung 3 ist entsprechend.

Von der Verklemmungsfreiheit kann man sich wie folgt überzeugen: Da es nur ein Objekt im Programm gibt, dessen Monitor gesperrt wird, kann eine mögliche Verklemmung nur daraus resultieren, dass Erzeuger und Verbraucher gegenseitig aufeinander warten. Es gilt aber folgende Invariante:

1. Entweder wartet kein Thread;
2. oder es wartet nur der Erzeuger, und der Puffer ist voll;
3. oder es wartet nur der Verbraucher, und der Puffer ist leer.

Diese Invariante gilt offensichtlich am Anfang der Programmausführung und wird von jedem möglichen Zustandsübergang erhalten, wie man sich relativ leicht überzeugt. Wir betrachten hier nur den bedingten Zustandsübergang am Anfang der Methode `ablegen`: Zum Zeitpunkt, zu dem der Erzeuger diesen Übergang vollziehen will, kann er nicht warten. Er ist ja gerade am Anfang der Ausführung der Methode `ablegen` und zwar bei der Auswertung des `if`-Ausdrucks `istVoll`. Die Invariante garantiert deshalb, dass entweder Fall (1) oder Fall (3) gilt.

Im Fall (1) gilt nach dem Zustandsübergang

- bei vollem Puffer Fall (2) (der Erzeuger ist durch `wait` in den Wartezustand gegangen; der Puffer ist voll und der Verbraucher muss nicht warten, wenn er die Methode `holen` ausführen will),
- bei nicht vollem Puffer weiterhin Fall (1) (der Erzeuger kann ablegen ohne zu warten. Der Verbraucher wartet zu dem Zeitpunkt auch nicht. Er hat vorher nicht gewartet und wartet daher auch im Moment des Zustandsübergangs des Erzeugers nicht. Der Verbraucher konnte seinen Zustand ja gar nicht ändern, da er nicht im Zustand rechnend war.)

Im Fall (3) ist der Puffer leer, also gilt nach dem Zustandsübergang weiterhin Fall (3). Der Erzeuger muss nicht warten, wohl aber der Verbraucher und zwar mindestens noch so lange, bis der Erzeuger die Methode ablegen beendet hat.

**Anwendungsmuster für die Methode `wait`.** Die oben behandelte Implementierung des Ringpuffers wird inkorrekt, wenn wir mehrere Erzeuger und Verbraucher zulassen. Beispielsweise kann Folgendes passieren: Der Puffer ist voll, und Erzeuger E1 geht in den Wartezustand. Ein Verbraucher holt ein Produkt aus dem Puffer und setzt E1 wieder rechenbereit. Bevor E1 allerdings weiter rechnen kann, lässt der Scheduler einen zweiten Erzeuger rechnen; dieser macht den Puffer wieder voll. Setzt anschließend E1 seine Ausführung fort, beendet er den `wait`-Aufruf bei vollem Puffer und führt mit der Speicherung des Produkts zu einem inkonsistenten Zustand. Dieses inkorrekte Verhalten resultiert daraus, dass die Bedingung, auf die im Beispiel gewartet wird, nach Beendigung des Wartezustands nicht automatisch erfüllt ist. Im Unterschied zu anderen Monitorrealisierungen, bei denen der Wartezustand immer mit einer bestimmten Bedingung verknüpft ist, ist der `wait/notify`-Mechanismus in Java nur eine Möglichkeit, die Ausführung eines synchronisierten Bereichs zu unterbrechen und den Bereich dabei freizugeben. Im Allgemeinen muss deshalb in Java nach Beendigung des Wartezustands geprüft werden, ob die Anforderungen für die weitere Ausführung des synchronisierten Bereichs erfüllt sind. Typischerweise verlegt man dazu den `wait`-Aufruf in eine Schleife, deren Bedingung der Wartebedingung entspricht. Wir demonstrieren dieses Anwendungsmuster am Beispiel der Methode `ablegen` aus der obigen `RingPuffer`-Klasse:

```
public synchronized void ablegen( Produkt prd )
                                throws InterruptedException {
    while( istVoll() ) wait();
    abspeichern( prd );
    notifyAll();                // Alle benachrichtigen!!!
}
```

Modifiziert man auch die Methode `holen` in entsprechender Weise, erhält man einen Ringpuffer, der eine beliebige Anzahl von Erzeugern und Verbrauchern verkraften kann. Die oben beschriebene Inkonsistenz kann nicht mehr auftreten, da die Bedingung an den Puffer nach Verlassen des `wait`-Aufrufs nochmals geprüft wird.

In den Zugriffsmethoden des Ringpuffers für mehrere Erzeuger und Verbraucher musste außerdem der `notify`-Aufruf durch einen `notifyAll`-Aufruf ersetzt werden. Andernfalls könnte es zu Verklemmungen kommen, wie das folgende Beispiel zeigt:

Angenommen, wir beließen es beim `notify`-Aufruf, der Ringpuffer hätte 4 Lagerplätze und es gäbe 8 Erzeuger und genau einen Verbraucher. Ein Erzeuger füllt den Puffer und wird dann in den Wartezustand versetzt; weitere 7 Erzeuger wollen ihre Produkte ablegen und müssen warten; damit sind 8 Erzeuger in der Wartemenge. Der Verbraucher leert den Puffer, setzt dabei insgesamt 4 der wartenden Erzeuger wieder rechenbereit und muss dann notgedrungen warten. Der erste der 4 rechenbereiten Erzeuger füllt anschließend den Puffer. Dabei wird viermal die Methode `notify` aufgerufen. Wenn diese Aufrufe die anderen 4 wartenden Erzeuger rechenbereit setzen, aber nicht den Verbraucher, gerät man in eine Situation, in der der Puffer voll ist und der Verbraucher wartet. Schließlich werden die Erzeuger der Reihe nach in den Wartezustand geraten.

Von der Verklemmungsfreiheit in der obigen Variante mit `notifyAll` überzeugt man sich auf die gleiche Weise wie beim Erzeuger-Verbraucher-Problem mit genau einem Erzeuger und genau einem Verbraucher; allerdings benötigt man dabei die folgende allgemeinere Invariante:

1. Entweder wartet kein Thread;
2. oder es warten nur Erzeuger, und der Puffer ist voll;
3. oder es warten nur Verbraucher, und der Puffer ist leer.

### 6.2.3 Zusammenfassung der sprachlichen Umsetzung von lokaler Parallelität

Dieser Abschnitt hat die wesentlichen Sprachkonzepte und -Konstrukte erläutert, die Java für die Implementierung lokal paralleler Programme bereitstellt. Der Programmierer kann mehrere nebenläufige Ausführungsstränge starten, die über gemeinsame Variablen kommunizieren können. Er besitzt Synchronisationskonstrukte, mit denen er die gleichzeitige Ausführung von Blöcken und Methoden durch mehrere Threads verhindern kann. Insofern ist lokale Parallelität in ähnlicher Weise sprachlich umgesetzt worden wie in üblichen prozeduralen, deklarativen oder anderen objektorientierten Sprachen<sup>10</sup>.

Insbesondere sind fast alle Objekte in einem Java-Programm *passiv*. Nur die Thread-Objekte sind aktiv in dem Sinne, dass die Ausführungsstränge, die durch ihre `run`-Methode festgelegt sind, (quasi-)parallel zu anderen Ausführungssträngen ausgeführt werden können. Von einer Umsetzung des objektorientierten Grundmodells kann man in Java also sicher nicht reden.

---

<sup>10</sup>Dass man Ausführungsstränge in mehreren anderen Sprachen „Prozesse“ nennt (z.B. in Smalltalk), sollte nicht darüber hinweg täuschen, dass es sich konzeptionell um gleichartige Realisierungstechniken handelt.

Im Wesentlichen gibt es zwei Aspekte, bei denen die Objektorientierung zum Tragen kommt:

1. Die Ausführungsstränge werden durch Objekte repräsentiert. Dadurch kann man sie recht gut vom Benutzerprogramm aus steuern.
2. Die Synchronisationsmechanismen sind auf Klassen und Objekte abgestimmt. Einerseits vereinfacht das die Objekt-spezifische Synchronisation und erlaubt ein engeres Zusammenspiel zwischen Kapselungs- und Synchronisationstechniken (siehe [Lea97] für eine weitergehende Behandlung dieser Aspekte). Andererseits verkompliziert es die Bedeutung der Synchronisationskonstrukte und erschwert damit deren korrekte Anwendung sowie Korrektheitsbeweise.



## Selbsttestaufgaben

### Aufgabe 1: Reisebüros

Das folgende JAVA-Programm beschreibt zwei Reisebüros, die parallel Tickets desselben Anbieters verkaufen.

```
class TicketAnbieter {
    private int VerfuegbareTickets;

    TicketAnbieter (int n) {
        VerfuegbareTickets = n;
    }
    boolean TicketsVerfuegbar() {
        return VerfuegbareTickets > 0;
    }
    int TicketVerkauf() {
        int nr = VerfuegbareTickets;
        VerfuegbareTickets = VerfuegbareTickets - 1;
        return nr;
    }
}

class Reisebuero extends Thread {
    private String name;
    private TicketAnbieter anbieter;

    Reisebuero(String name, TicketAnbieter anbieter) {
        this.name = name;
        this.anbieter = anbieter;
    }
    void warteAufKunde() {
        try {Thread.sleep(Math.round( 1000 * Math.random() ));}
        catch (InterruptedException e) {}
    }
    public void run() {
        warteAufKunde();
        while (anbieter.TicketsVerfuegbar()) {
            int nr = anbieter.TicketVerkauf();
            System.out.println( name + " verkauft Ticket " + nr);
            warteAufKunde();
        }
    }
}

class Test {
    public static void main(String [] argv) {
        TicketAnbieter ta = new TicketAnbieter(4);
        new Reisebuero("Reiseland", ta).start();
        new Reisebuero("Happy Travel", ta).start();
    }
}
```

- a) Anstelle Subklassen von der Thread-Klasse zu bilden, kann man Threads in JAVA auch mit Hilfe der Schnittstelle `Runnable` implementieren. Ändern Sie das obige Programm so ab, dass diese Technik verwendet wird.
- b) Beschreiben Sie einen Ablauf des Programms, bei dem beide Reisebüros das Ticket mit der Nummer 3 verkaufen, bei dem also sowohl Happy Travel verkauft Ticket 3 als auch Reiseland verkauft Ticket 3 ausgegeben wird.
- c) Um das in b) beschriebene Problem zu lösen, vereinbart ein Kollege von Ihnen die Methoden der Klasse `TicketAnbieter` als `synchronized`:

```
synchronized boolean TicketsVerfuegbar() { ... }  
  
synchronized int TicketVerkauf() { ... }
```

Leider ist dies aber immer noch keine befriedigende Lösung, denn es kann passieren, dass das (nicht-existente) Ticket 0 verkauft wird, d.h. es kann z.B. Reiseland verkauft Ticket 0 ausgegeben werden. Beschreiben Sie einen Ablauf, der zu dieser Ausgabe führt.

- d) Lösen Sie das in c) angesprochene Problem, indem Sie das Programm geeignet abändern. Dabei sollen sich die beiden Threads möglichst wenig gegenseitig behindern.

**Aufgabe 2: Wertetausch**

Gegeben sei das folgende Java-Programm:

```
class Wert {
    int wert;

    Wert(int wert) {
        this.wert = wert;
    }
}

class Tausche extends Thread {
    Wert a;
    Wert b;

    Tausche(Wert a, Wert b) {
        this.a = a;
        this.b = b;
    }

    public void run() {
        int h = a.wert;
        a.wert = b.wert;
        b.wert = h;
    }
}

class Main {
    public static void main(String[] argv) {
        Wert x = new Wert(0);
        Wert y = new Wert(1);
        Tausche tom = new Tausche(x,y);
        Tausche jerry = new Tausche(y,x);
        tom.start();
        jerry.start();
    }
}
```

- a) Programmierer D. Ummbatz, der Verfasser des oben angegebenen Programms, hatte sich gedacht, dass die von `x` und `y` referenzierten Objekte wieder ihre Initialwerte enthalten, wenn die beiden Tausche-Threads ihre `run`-Methode beendet haben. Er argumentierte so:

Der eine Thread vertauscht die Werte und der andere Thread tauscht sie dann wieder zurück.

Zeigen Sie, dass diese Argumentation falsch ist. Beschreiben Sie dazu einen Ablauf des Programms, in dem die von `x` und `y` referenzierten Objekte beide den Wert 0 enthalten, nachdem die beiden Threads ihre `run`-Methode vollständig abgearbeitet haben.



- b) Um dieses Problem zu beseitigen, verändert Programmierer D. Ummatz die `run`-Methode wie folgt:

```
public void run() {  
    synchronized (a) {  
        synchronized (b) {  
            int h = a.wert;  
            a.wert = b.wert;  
            b.wert = h;  
        }  
    }  
}
```

Beschreiben Sie einen Ablauf des so veränderten Programms, bei dem eine Verklemmung (*deadlock*) auftritt.

- c) Ergänzen Sie die Klasse `Tausche` um Codestücke, so dass wechselseiger Ausschluss der beiden Vertauschungsvorgänge gewährleistet ist und keine Verklemmungen auftreten können. Verändern Sie dabei die Klassen `Wert` und `Main` nicht. Begründen Sie, warum Ihre Lösung das Gewünschte leistet.

### Aufgabe 3: Druckerbenutzung

Das folgende JAVA-Programm beschreibt zwei Benutzer Hänsel und Gretel, die denselben Drucker benutzen. Jeder der Benutzer druckt eine Datei mehrfach (genauer gesagt fünfmal) auf dem Drucker aus. Die Namen der zu druckenden Dateien werden der `main`-Methode der Klasse `Test` als Parameter übergeben. Im Folgenden bezeichnen wir das einmalige Drucken einer Datei als einen *Druckjob*. Jeder Benutzer initiiert also fünf Druckjobs.

```
import java.io.*;

class Drucker {
    void druckeDatei(String dateiname) {
        try {
            BufferedReader in = new BufferedReader(new FileReader(dateiname));
            String line = in.readLine();
            while (line != null) {
                // Zeile line auf dem Drucker ausgeben
                ...
                line = in.readLine();
            }
        }
        catch (Exception e) {
            System.out.println("Eine Ausnahme ist aufgetreten.");
        }
    }
}

class Benutzer extends Thread {
    Drucker drucker;
    String dateiname;
    int anzahl;

    Benutzer(Drucker drucker, String dateiname, int anzahl) {
        this.drucker = drucker;
        this.dateiname = dateiname;
        this.anzahl = anzahl;
    }

    public void run() {
        for (int i=0; i<anzahl; i++) {
            drucker.druckeDatei(dateiname);
        }
    }
}
```

```

class Test {
    public static void main (String[] argv) {
        if (argv.length >= 2) {
            Drucker d = new Drucker();
            Benutzer haensel = new Benutzer(d, argv[0], 5);
            Benutzer gretel = new Benutzer(d, argv[1], 5);
            haensel.start();
            gretel.start();
        }
        else
            System.out.println("Bitte zwei Dateinamen als Argumente uebergeben!");
    }
}

```

- a) Bei dem angegebenen Programm kann es passieren, dass ein Druckjob von Hänsel mit einem Druckjob von Gretel vermischt auf dem Drucker ausgegeben wird. Beschreiben Sie einen Programmlauf, bei dem das passiert.
- b) Verändern Sie das Programm so, dass sich Druckjobs nicht mehr miteinander vermischen können. Ihre Lösung soll aber nicht so restriktiv sein, dass alle Druckjobs eines Benutzers als Einheit behandelt werden. Zum Beispiel soll es möglich sein, dass erst die Datei von Hänsel einmal gedruckt wird, dann die von Gretel, dann wieder die von Hänsel usw. Begründen Sie, warum Ihr Vorschlag das Problem löst.
- c) Synchronisieren Sie die beiden Threads für Hänsel und Gretel nun so, dass zunächst alle Druckjobs des einen Benutzers gedruckt werden und erst im Anschluss daran die Druckjobs des anderen Benutzers. Begründen Sie, warum Ihr Vorschlag das Problem löst.

#### Aufgabe 4: Automatische Zählerklasse

In dieser Aufgabe sollen Sie Java's Unterbrechungsmechanismus üben. Dazu soll die in Kurseinheit 5, Aufgabe 3, vorgestellte Benutzungsoberfläche für die Klasse `Counter` wie folgt geändert werden. Der Zählerstand soll nicht mehr mit Hilfe der Buttons „<“ und „>“ verändert werden, sondern die Benutzungsoberfläche soll einen eigenen Thread starten, der den Zählerstand alle 500 Millisekunden um eins erhöht und im zugehörigen Textfeld anzeigt. Die Buttons zum Erhöhen und Erniedrigen des Zählerstandes entfallen in dieser Aufgabe. Bei Betätigung der mit „quit“ beschrifteten Schaltfläche soll durch einen Aufruf von `interrupt` eine Unterbrechungsanforderung für den Zähl-Thread erzeugt werden. Dieser Thread soll dann den aktuellen Zählerstand fertig ausgeben und anschließend terminieren. Dabei soll der Zähl-Thread seine `run`-Methode nur auf Grund des Ergebnisses der Methode `isInterrupted` beenden.

## Musterlösungen zu den Selbsttestaufgaben

### Aufgabe 1: Reisebüros

- a) Die erste Zeile der Klassendefinition von `Reisebuero` muss wie folgt abgeändert werden:

```
class Reisebuero implements Runnable {
```

Ferner müssen die Anweisungen, die die beiden Threads erzeugen, verändert werden, also die letzten beiden Zeilen der `main`-Methode der Klasse `Test`:

```
new Thread(new Reisebuero("Reiseland", ta)).start();
new Thread(new Reisebuero("Happy Travel", ta)).start();
```

- b) Das Ticket 3 kann z.B. durch den folgenden Ablauf von beiden Reisebüros verkauft werden: Bevor der `Happy Travel`-Thread Rechenzeit zugeteilt bekommt, macht der `Reiseland`-Thread einen vollen Durchlauf durch den Schleifenrumpf seiner `run`-Methode und verkauft dabei das Ticket 4 (Ausgabe `Reiseland verkauft Ticket 4`). Dann beginnt er einen zweiten Schleifendurchlauf, wird dabei aber zwischen den beiden Anweisungen

```
int nr = VerfuegbareTickets;
```

und

```
VerfuegbareTickets = VerfuegbareTickets - 1;
```

in der Methode `anbieter.TicketVerkauf` unterbrochen. Zu diesem Zeitpunkt hat das Attribut `VerfuegbareTickets` noch den Wert 3 aber in der lokalen Variablen `nr` ist bereits gespeichert, dass die Methode `TicketVerkauf` später 3 zurückgeben wird. Nun führt der `Happy Travel`-Thread den Schleifenrumpf seiner `run`-Methode einmal aus und gibt dabei `Happy Travel` verkauft Ticket 3 aus. Wenn dann später der `Reiseland`-Thread wieder Rechenzeit zugeteilt erhält, druckt er `Reiseland verkauft Ticket 3`, verkauft also ebenfalls das Ticket 3.

- c) Das Problem rührt daher, dass die Threads zwischen der Abfrage, ob noch ein Ticket verfügbar ist, und dem Kauf dieses Tickets vom Ticketanbieter unterbrochen werden können. Während der Unterbrechung, kann der andere Thread alle verfügbaren Tickets verkaufen. Wird der erste Thread dann wieder aktiv, glaubt er irrigerweise, es wäre noch ein Ticket verfügbar und verkauft das nicht-existente Ticket 0.

- d) Es muss verhindert werden, dass ein Thread zwischen der Abfrage, ob noch ein Ticket verfügbar ist, und dem Kauf dieses Tickets unterbrochen wird. Im Folgenden beschreiben wir eine Lösung, bei dem sich das Reisebüro vor dem Ticketkauf noch einmal rückversichert, ob noch ein Ticket verfügbar ist. Um zu erreichen, dass zwischen der Rückversicherung und dem Ticketverkauf kein anderer Thread Methoden des Ticketanbieters ausführen kann, vereinbart man alle Methoden des Ticketanbieters als `synchronized` und führt Rückversicherung und Ticketverkauf in einem `synchronized`-Block mit dem Monitor des Ticketanbieters aus:

```
public void run() {
    warteAufKunde();
    while (anbieter.TicketsVerfuegbar()) {
        synchronized (anbieter) {
            if (anbieter.TicketsVerfuegbar()) {
                int nr = anbieter.TicketVerkauf();
                System.out.println( name + " verkauft Ticket " + nr);
            }
        }
        warteAufKunde();
    }
}
```

Im Rahmen des gegebenen Programms kann man sogar darauf verzichten, die Methoden des Ticketanbieters als `synchronized` zu vereinbaren, da beide Reisebüros sich sowieso an das „Protokoll“ halten, Tickets nur dann vom Ticketanbieter zu kaufen, wenn sie im Besitz seines Monitors sind. In der „realen Welt“ könnte es aber Reisebüros geben, die sich nicht an dieses Protokoll halten.

**Aufgabe 2: Wertetausch**

- a) Durch die Reihenfolge der Argumente in den Konstruktoraufrufen der beiden Tausche-Objekte `tom` und `jerry` verweisen die Attribute `tom.a` und `jerry.b` auf dasselbe Objekt wie `x` und `tom.b` und `jerry.a` auf dasselbe Objekt wie `y`. Im Folgenden beschreiben wir einen Ablauf, nach dem beide Objekte den Wert 0 enthalten.

Der `tom`-Thread beginnt zu laufen, wird aber nach der ersten Zuweisung in seiner `run`-Methode, `int h:=a.wert`, unterbrochen. Nun beginnt der `jerry`-Thread zu laufen, führt alle drei Zuweisungen in seiner `run`-Methode durch und terminiert. Schliesslich führt der `tom`-Thread seine noch verbleibenden zwei Zuweisungen aus. Die folgende Tabelle zeigt, dass nach diesem Ablauf tatsächlich in beiden Objekten `x` und `y` der Wert 0 gespeichert ist. Die ersten beiden Spalten enthalten die von den beiden Threads ausgeführten Zuweisungen in der oben beschriebenen Reihenfolge. In den letzten vier Spalten vermerken wir jeweils den Wert, der durch diese Zuweisung verändert wird.

tom	jerry	Attrib. wert in x, tom.a und jerry.b	Attrib. wert in y, tom.b und jerry.a	lok. Var. h in run von tom	lok. Var. h in run von jerry
		0	1		
<code>int h = a.wert</code>				0	
	<code>int h = a.wert</code>				1
	<code>a.wert = b.wert</code>		0		
	<code>b.wert = h</code>	1			
<code>a.wert = b.wert</code>		0			
<code>b.wert = h</code>			0		

Wie man sieht, enthalten die `wert`-Attribute von `x` und `y` bei Terminierung beide den Wert 0.

- b) Im Folgenden beschreiben wir einen Ablauf des wie in der Aufgabenstellung veränderten Programms, bei dem eine Verklemmung auftritt.

Nach der Generierung der `Wert`-Objekte `x` und `y` und der beiden Threads `tom` und `jerry` beginnt `tom` mit der Ausführung seiner `run`-Methode. Er betritt den Monitor des von seinem Attribut `a` referenziereten Objekts (also den von `x`) und wird dann unterbrochen. Nun beginnt `jerry` mit der Ausführung seiner `run`-Methode. Da wegen der vertauschten Reihenfolge der Argumente in den Konstruktoraufrufen für `tom` und `jerry` das Attribut `a` von `jerry` auf das Objekt `y` verweist, dessen Monitor noch nicht gesperrt ist, kann `jerry` den Monitor von `y`

betreten und tut dies auch. In dieser Situation möchte `jerry` den Monitor von `x` betreten, der aber von `tom` gesperrt ist. Andererseits möchte `tom` den Monitor von `y` betreten, der aber von `jerry` gesperrt ist, eine typische Verklemmungssituation.

- c) Hier ist eine Modifikation der Klasse `Tausche`, die wechselweisen Ausschluss garantiert:

```
class Tausche extends Thread {
    Wert a;
    Wert b;

    static Object lock = new Object();

    Tausche(Wert a, Wert b) {
        this.a = a;
        this.b = b;
    }

    public void run() {
        synchronized(lock) {
            int h = a.wert;
            a.wert = b.wert;
            b.wert = h;
        }
    }
}
```

Die Klasse `Tausche` wurde um ein statisches Attribut `lock` erweitert, das mit einem Wert vom Typ `Object` initialisiert wurde. Der Code, der die Vertauschung der Werte vornimmt, wurde mit Hilfe eines `synchronized`-Blocks in den Monitor des vom Attribut `lock` referenzierten Objekts verlagert.

Diese Lösung garantiert wechselweisen Ausschluss: Da das Attribut `lock` als statisches Attribut vereinbart wurde, verwenden beide Threads dasselbe Objekt zur Synchronisation. Wenn der erste Thread mit seinem Vertauschungsvorgang beginnt, hat er den Zugang zu dem Monitor dieses Objekts zuvor für die gesamte Dauer der Vertauschung gesperrt. Daher kann der andere Thread erst dann mit dem Vertauschen beginnen, wenn der erste Thread seine Vertauschung vollständig durchgeführt hat.

Verklemmungen können bei dieser Lösung nicht auftreten, da nur ein einziger Monitor gesperrt wird.

Beachten Sie, dass es nicht genügt, die `run`-Methode der Klasse `Tausche` als `synchronized` zu vereinbaren. Zwar müsste jeder der

Threads dann vor Ausführung seiner `run`-Methode einen Monitor betreten. Da aber jeder der beiden `Tausche`-Objekte seinen eigenen Monitor besitzt, würden die beiden Threads verschiedene Monitore betreten, so dass wechselseitiger Ausschluss nicht gewährleistet wäre. Aus einem ähnlichen Grunde funktioniert die oben angegebenen Lösung auch nur, weil das Attribut `lock` als statisch vereinbart wurde, so dass beide Threads denselben Monitor (nämlich den von `lock`) betreten müssen.

### Aufgabe 3: Druckerbenutzung

- a) Es folgt eine Beschreibung eines derartigen Szenarios. Der Hänsel-Thread beginnt durch den Aufruf von `drucker.druckeDatei` in der Schleife in seiner `run`-Methode, seine Datei zum ersten Mal zu drucken. Nachdem er einige, aber noch nicht alle Zeilen dieser Datei gedruckt hat, d.h. nach dem ersten, aber vor dem letzten Durchlauf der Schleife im Rumpf von `druckeDatei`, wird er unterbrochen und dem Gretel-Thread wird Rechenzeit zugeteilt. Nun kann der Gretel-Thread seinerseits durch den Aufruf von `drucker.druckeDatei` in der Schleife in seiner `run`-Methode beginnen, seine Datei zu drucken. Damit vermischt sich der erste Ausdruck der Datei von Hänsel mit dem ersten Ausdruck der Datei von Gretel.
- b) Eine einfache Lösung, das Geforderte zu erreichen, ist, die `druckeDatei`-Methode als synchronisierte Methode zu vereinbaren, also ihren Kopf wie folgt zu verändern:

```
synchronized void druckeDatei(String dateiname) {
```

Dadurch sperrt ein Thread den Monitor des `Drucker`-Objekts, bevor er mit dem Drucken eines seiner Druckjobs beginnt. Der Monitor bleibt während des gesamten Druckvorgangs für diesen Druckjob gesperrt. Der andere Thread muss aber zum Drucken seines Druckjobs ebenfalls den Monitor des `Drucker`-Objekts betreten. Deswegen kann er erst dann mit dem Drucken seines nächsten Druckjobs beginnen, wenn der Monitor wieder freigegeben ist, also nachdem der erste Thread seinen Druckjob vollständig gedruckt hat. Deshalb können sich Druckjobs nicht mehr vermischen.

- c) Eine Lösung, die das Geforderte leistet, ist, zusätzlich zur Synchronisierung der Methode `druckeDatei` die Schleife in der `run`-Methode durch den Monitor des Druckers überwachen zu lassen. Dies kann durch einen `synchronized`-Block leicht realisiert werden:



```

public void run() {
    synchronized(drucker) {
        for (int i = 0; i < anzahl; i++) {
            drucker.druckeDatei(dateiname);
        }
    }
}

```

Damit sichert sich der Benutzer exklusiven Zugang zum Drucker für alle seine Druckjobs. Denn während er die Schleife ausführt, in der die Druckjobs abgesetzt werden, ist der Monitor des Druckers gesperrt, so dass kein anderer Benutzer den Rumpf der Methode `druckeDatei` ausführen kann. Im Rahmen des gegebenen Programms kann man auf die Synchronisierung der Methode `druckeDatei` verzichten, da sich die beiden Benutzer Hänsel und Gretel an das Protokoll halten, den Monitor des Druckers vor dem Initiieren ihrer Druckjobs zu sperren. Wenn es aber noch andere Benutzer geben könnte, die den Drucker benutzen, muss die Methode `druckeDatei` synchronisiert werden, da die anderen Benutzer dieses Protokoll verletzen könnten.

#### Aufgabe 4: Automatische Zählerklasse

Folgendes Programm löst die gestellte Aufgabe. Der erneute Aufruf von `interrupt` in der mit (+) gekennzeichneten Zeile ist nötig, da beim Werfen der Ausnahme `InterruptedException` das Unterbrechungsflag zurück gesetzt wird, das durch `isInterrupted` abgefragt wird. Da der Zähl-Thread aber nur auf Grund des Ergebnisses von `isInterrupted` beendet werden sollte, mußte das Flag neu gesetzt werden. Ansonsten hätte man die `while`-Schleife innerhalb von `run` auch direkt mit der `break`-Anweisung beenden können.

```

import java.awt.*;
import java.awt.event.*;

class Counter {

    private int startValue;
    private int value;
    private int min = 0;
    private int max = 99;

    Counter (int start) {
        startValue = start;
        value = start;
    }

    void incr() { if (value < max) value++; }
    void decr() { if (value > min) value--; }
}

```

```

    void resetCounter () {value = startValue;}
    public String toString() { return "" + value; }
}

public class CountFrame extends Frame {

    Button bQuit = new Button ("quit");
    TextField tCounter = new TextField(3);
    Counter c = new Counter (50);
    CounterThread cth = new CounterThread();

    // Zaehl-Thread
    private class CounterThread extends Thread {

        Counter counter;

        public void run() {
            while (true) {
                if (isInterrupted()) {
                    break;
                }
                c.incr();
                tCounter.setText(c.toString());

                // 500 Millisekunden warten
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    (+) interrupt();
                }
            }
        }
    }

    public CountFrame () {

        tCounter.setEditable(false);
        tCounter.setText(c.toString());

        // geeignete Beobachter registrieren
        bQuit.addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                cth.interrupt();
            }
        });
        addWindowListener (new WindowAdapter() {
            public void windowClosing (WindowEvent e) {
                System.exit(0);
            }
        });
    }
}

```

```
// Textfeld in Panel einfuegen
Panel p = new Panel(new FlowLayout());
p.add (tCounter);

// Komponenten in Hauptfenster einfuegen
add (p, BorderLayout.CENTER);
add (bQuit, BorderLayout.SOUTH);

// Fenstergroesse festlegen
setSize (120, 100);

setVisible(true);

// Zaehl-Thread starten
cth.start();
}

public static void main(String[] args) {
    CountFrame countFrame = new CountFrame();
}
}
```

# Studierhinweise zur Kurseinheit 7

Diese Kurseinheit umfasst die Kapitel 7 und 8 des Kurstextes. Es wird von Ihnen erwartet, dass Sie die Abschnitte 7.1 und 7.2 im Detail studieren und verstehen. Den Rest der Kurseinheit sollten Sie so weit durcharbeiten, dass Sie die dargestellten Konzepte grundsätzlich begriffen haben und entsprechende Fragen beantworten können. Prüfen Sie nach dem Durcharbeiten des Kurstextes, ob Sie die Lernziele erreicht haben. Arbeiten Sie zur Überprüfung auch wieder die Aufgaben am Ende der Kurseinheit durch.

## **Lernziele:**

- Grundlegende Aspekte verteilter Systeme.
- Genaues Verständnis der Client-Server-Architektur. (Was unterscheidet Clients von Servern? Was ist der Vorteil einer solchen unsymmetrischen Kommunikation? Wie bedient ein Server mehrere Clients gleichzeitig?)
- Kommunikation von Client und Server über Socket-Verbindungen und ihre Realisierung in Java.
- Grundsätzliche Kenntnis der Client-Server-Architektur des WWW. (Wie sieht die Clientseite, wie die Serverseite aus? Was leistet der Server? Was ist ein URL? Was ist HTML? Wie kommunizieren Clients und Server?)
- Object-Request-Broker-Architektur.
- Realisierung der Kommunikation zwischen verteilten Objekten mittels Aufruf entfernter Methoden (RMI) in Java.
- Objektorientierte Grund- und Sprachkonzepte.
- Aspekte anderer objektorientierter Sprachen im Vergleich zu Java.
- Zukünftige Entwicklungslinien.

Bitte vergessen Sie nicht, uns Ihre Kritik – negative wie positive – und Korrekturen am Kurs mitzuteilen. Im Voraus vielen Dank dafür, insbesondere auch im Namen Ihrer Kommilitonen, die den Kurs in Zukunft belegen!

# Kapitel 7

## Programmierung verteilter Objekte

Dieses Kapitel hat die Programmierung verteilter objektorientierter Systeme als Thema. Der Schwerpunkt liegt dabei auf Client-Server-Kommunikation und deren Realisierung in Java. Der erste Abschnitt bietet eine kurze Einführung in zentrale Aspekte und Problemstellungen verteilter objektorientierter Systeme. Der zweite Abschnitt behandelt die Kommunikation von Programmen über ein Netzwerk mit Hilfe sogenannter Sockets. Der dritte Abschnitt erläutert, was entfernte Methodenaufrufe sind und wie sie für die verteilte objektorientierte Programmierung genutzt werden können. Wir werden uns hier auf die elementaren Mechanismen beschränken. Bezüglich weiterführender Aspekte, insbesondere der Sicherheitsproblematik, verweisen wir auf die Literatur (siehe z.B. [Far98]).

### 7.1 Verteilte objektorientierte Systeme

Dieser Abschnitt fasst zunächst grundlegende Aspekte verteilter Systeme zusammen und geht danach auf spezifische Problemstellungen im Rahmen objektorientierter verteilter Systeme ein.

#### 7.1.1 Grundlegende Aspekte verteilter Systeme

Unter einem *verteilten System* verstehen wir eine Menge von lose gekoppelten Prozessen<sup>1</sup>, d.h. von Prozessen, die unabhängig voneinander Berechnungen ausführen und miteinander kommunizieren können (vgl. Abschn. 6.1.1, S. 378). Jeder Prozess arbeitet dabei ein eigenes Programm in einem eigenen Adressraum ab; insbesondere arbeiten die Prozesse im Allgemeinen parallel.

*verteiltes  
System*

Typischerweise sind verteilte Systeme räumlich, d.h. auf mehrere Rechner verteilt; aber wir betrachten auch eine Menge von lose gekoppelten Prozes-

---

<sup>1</sup>Häufig werden auch die Rechner, auf denen die Prozesse ausgeführt werden, als verteiltes System bezeichnet.

sen auf *einem* Rechner als verteiltes System, da die relevanten Fragestellungen und Probleme die gleichen sind. Eine Menge eng gekoppelter Prozesse (das sind Prozesse, die sich einen gemeinsamen Adressraum teilen) oder eine Menge von Threads betrachten wir nicht als verteiltes System. Drei Aspekte unterscheiden verteilte Systeme von sequentiellen oder eng gekoppelten parallelen Systemen:

1. Bei der Untersuchung verteilter Systeme steht die Kommunikation im Mittelpunkt der Betrachtung; insbesondere spielt die Zeit, die für die Kommunikation benötigt wird, bei der Analyse von verteilten Systemen eine wichtige Rolle.
2. Der Ausfall einzelner Prozesse/Rechner darf die Kommunikation anderer Prozesse/Rechner in einem verteilten System nicht beeinträchtigen; in diesem Zusammenhang sind Maßnahmen zur Fehlertoleranz wichtig.
3. Im Allgemeinen kann bei einem verteilten System nicht davon ausgegangen werden, dass der globale Systemzustand den beteiligten Prozessen bekannt ist. Diese haben nur eine eingeschränkte, lokale Sicht auf das Gesamtsystem; insbesondere können verteilte Systeme während der Systemlaufzeit erweitert werden, ohne dass alle Prozesse davon erfahren (man denke beispielsweise an das WWW, das kontinuierlich wächst).

Da die Kommunikation im Rahmen verteilter Systeme eine zentrale Rolle spielt, wollen wir uns ihre wichtigsten Aspekte genauer anschauen: die Art, wie die Kommunikation abläuft; die Mittel, mit denen die Kommunikation stattfindet; die Organisation von Kommunikation.

**Kommunikationsart.** Die Kommunikation zwischen zwei Prozessen A und B eines verteilten Systems kann *synchron* oder *asynchron* erfolgen.

*synchron*  
*asynchron*

Synchrone Kommunikation bedeutet, dass A und B nur dann kommunizieren können, wenn die Ausführung beider Prozesse zueinander passende Programmstellen erreicht hat; man spricht vom Rendezvous der Prozesse. Andernfalls muss ein Prozess warten, bis der andere auch kommunikationsbereit ist: Wenn beispielsweise Prozess A eine Programmstelle erreicht, an der er mit B kommunizieren will, und B gerade mit anderen, internen Berechnungen beschäftigt ist, muss A warten, bis B eine Programmstelle erreicht, an der auch er kommunizieren will. Erreicht B auch eine solche Programmstelle, findet die Kommunikation statt (z.B. Übergabe von Objekten, Nachrichten). Nach Ende der Kommunikation gehen beide Prozesse wieder ihrer Wege.

Asynchrone Kommunikation bedeutet, dass Prozess A die Aktionen für eine Kommunikation (z.B. das Verschicken einer Nachricht) ausführt und danach weiter arbeitet, ohne sich um B zu kümmern (entsprechend für Prozess

B). Asynchrone Kommunikation setzt in der Regel die Pufferung von Nachrichten voraus. Für die Synchronisation asynchron kommunizierender Prozesse werden zusätzliche Techniken und Sprachmittel eingesetzt (z.B. Monitore, Semaphore). Synchronisation ist z.B. nötig für den Zugriff auf einen gemeinsamen Nachrichtenpuffer oder sonstige gemeinsame Speicher.

**Kommunikationsmittel.** Im Wesentlichen kann man vier Kommunikationsmittel unterscheiden. Allerdings sind die Übergänge zwischen ihnen fließend und in der Praxis finden sich oft Kombinationen dieser Varianten:

1. Kommunikation über gemeinsamen Speicher: Der Begriff „Speicher“ ist hierbei weit zu fassen. Beispielsweise können damit Programmvariablen gemeint sein, auf die die kommunizierenden Prozesse gemeinsam Zugriff haben, oder auch komplexere Datenstrukturen, wie z.B. Ein- und Ausgabeströme.
2. Kommunikation über Nachrichten: Der Absender schickt eine Nachricht mit einer Empfängeradresse ab; ein Netzwerk stellt die Nachricht dem Empfänger zu; der Empfänger liest eingetroffene Nachrichten aus einem Puffer.
3. Entfernter Prozedur- bzw. Methodenaufruf: Die Kommunikation verläuft ähnlich wie bei einem lokalen Prozedur-/Methodenaufruf mit Parameterübergabe und Ergebniserückgabe.
4. Spezielle Kommunikationskonstrukte: Insbesondere für die synchrone Kommunikation werden spezielle Sprachkonstrukte eingesetzt, die die Kommunikationspunkte innerhalb der Prozesse und die Parameterübergabe beschreiben<sup>2</sup>.

Die ersten zwei Kommunikationsmittel werden hauptsächlich für asynchrone Kommunikation verwendet, die anderen beiden fast ausschließlich für synchrone Kommunikation.

**Organisation der Kommunikation.** Die Kommunikation zwischen Prozessen in einem verteilten System kann sehr unterschiedlich organisiert sein. Innerhalb eines Systems können Prozesse unterschiedliche Rollen übernehmen, sie können sich zu Gruppen zusammenschließen oder auch völlig gleichberechtigt agieren. Die verbreitetste Organisationsform sind die sogenannten *Client-Server-Architekturen*: Ein Server (z.B. HTTP-Server) ist ein

---

<sup>2</sup>Beispielsweise kann man zu einem Prozess P in Ada sogenannte Entries deklarieren. In den Anweisungen von P wird festgelegt, an welchen Stellen P bereit ist, Entry-Aufrufe zu akzeptieren und wie darauf reagiert werden soll. Andere Prozesse können diese Entries aufrufen und damit Parameter an P übergeben. Allerdings müssen sie mit der Parameterübergabe warten, bis P eine Stelle erreicht, an der P einen entsprechenden Entry-Aufruf akzeptiert.



Prozess, der Dienste für Client-Prozesse (z.B. Prozesse, die Internet-Browser ausführen) erbringt. Server laufen ununterbrochen. In der einfachsten Form warten sie auf eine Anforderung von einem Client (z.B. das Anfordern einer WWW-Seite); trifft eine Anforderung ein, wird diese bearbeitet und danach wird auf die nächste Anforderung gewartet. Um mehrere Clients gleichzeitig bedienen zu können, müssen Server eintreffende Anforderungen entweder puffern, oder sie müssen Unterprozesse mit der Erledigung von Client-Anforderungen betrauen. In bestimmten Client-Server-Architekturen sorgt der Server auch dafür, dass mehrere Clients miteinander kommunizieren können (typisches Beispiel dafür sind sogenannte Chat-Server).

*Request-  
Broker-  
Architektur*

Mehr Flexibilität als klassische Client-Server-Architekturen bieten sogenannte *Request-Broker-Architekturen*. Request-Broker stehen zwischen Client und Server und vermitteln Client-Anforderungen an einen geeigneten verfügbaren Server. Andere Organisationsformen in verteilten Systemen sind Gruppen-Architekturen, die das An- und Abmelden von Prozessen in Gruppen unterstützen und es ermöglichen, mit allen Mitgliedern der Gruppe zu kommunizieren bzw. sich mit allen Mitgliedern zu synchronisieren.

### 7.1.2 Programmierung verteilter objektorientierter Systeme

Zwei Eigenschaften prädestinieren das objektorientierte Grundmodell und damit objektorientierte Programmiersprachen und -techniken für die Programmierung verteilter Systeme (vgl. die Abschnitte 1.2.3 und 6.1.2):

1. Objekte fassen Daten und Operationen zu Einheiten mit klar definierten Schnittstellen zusammen; Objekte bzw. Klassen bilden demnach weitgehend unabhängige Programmteile, die sich auf verschiedene Prozesse bzw. Rechner verteilen lassen.
2. Kommunikation zwischen Objekten über Nachrichten ist integraler Bestandteil des objektorientierten Grundmodells, sodass bereits alle wesentlichen Sprachmittel für die Kommunikation im Rahmen verteilter Systeme zur Verfügung stehen<sup>3</sup>.

*homogen*

Trotz dieser guten Grundvoraussetzungen ergeben sich auch bei der Programmierung objektorientierter verteilter Systeme durch den Verteilungsaspekt zusätzliche konzeptionelle und softwaretechnische Fragestellungen. Um diese Fragestellungen zu erörtern, betrachten wir zunächst *homogene* verteilte Systeme. Das sind verteilte Systeme, bei denen die zentralen Systemparameter, wie z.B. die Betriebssystemschnittstelle, der einzelnen Teile gleich

<sup>3</sup>Hier fokussieren wir auf das objektorientierte Grundmodell. Wie wir schon gesehen haben, bieten viele OO-Sprachen keinen wirklichen Nachrichtenaustausch über z.B. Queues. Der „Nachrichtenaustausch“ besteht i.a. im Aufruf von Methoden auf dem „Empfängerobjekt“.

sind. Insbesondere gehen wir davon aus, dass alle Teile homogener Systeme in der gleichen Programmiersprache verfasst sind (z.B. in Java). Danach gehen wir kurz auf die zusätzlichen Fragestellungen ein, die in *heterogenen* verteilten Systemen zu beantworten sind. *heterogen*

**Homogene Systeme.** Wir betrachten ein einfaches verteiltes System bestehend aus zwei Prozessen A und B, verteilt auf zwei Rechner, und nehmen an, dass die von A und B ausgeführten Programme in der gleichen Programmiersprache geschrieben sind. Das objektorientierte Grundmodell legt nahe, dass A und B mittels des Methodenaufrufmechanismus kommunizieren. Dabei stellt sich die Frage, wie ein Objekt, das in Prozess A existiert, eine Referenz auf ein Objekt in B erhalten und abspeichern kann.

Üblicherweise wird dieses Problem wie folgt gelöst: Jedes Objekt, das B bereitstellen möchte, trägt B unter einem Namen in eine Tabelle ein. Diese Tabelle wird von einem eigenen Prozess verwaltet, den man meist Namensserver oder Registry nennt. Bei dem Namensserver kann sich Prozess A über den Namen des gesuchten Objekts dessen Referenz besorgen und damit dann Methoden des B-Objekts aufrufen. In einer typisierten Sprache muss A dazu die öffentliche Schnittstelle des gesuchten Objekts bekannt sein; eine dynamische Prüfung sollte feststellen, ob die erhaltene Referenz auch auf ein Objekt des in A erwarteten Typs zeigt.

Ein geeigneter Mechanismus in Prozess B muss dann dafür sorgen, dass die Methoden, die von Prozess A aus aufgerufen werden, ausgeführt werden und das Ergebnis an den Aufrufer in A zurückgeschickt wird. Für den Fall, dass mehrere Prozesse im verteilten System existieren, kann es dabei, wie bei lokaler Parallelität, dazu kommen, dass eine Methode gleichzeitig von verschiedener Seite aufgerufen wird, sodass die in Kap. 6 angesprochenen Konflikte auftreten können.

Außer dem oben skizzierten Aspekt, dass Referenzen auf entfernte Objekte über explizite Namen angesprochen werden müssen, gibt es keinen konzeptionellen Unterschied zwischen lokalen und verteilten objektorientierten Systemen. Softwaretechnische Gründe führen allerdings dazu, lokale und entfernte Objekte in weiteren Aspekten unterschiedlich zu behandeln. Da in gängigen Anwendungen der entfernte Methodenaufruf verglichen mit dem lokalen Methodenaufruf relativ selten ist, ist es sinnvoll, nur diejenigen Objekte mit dem notwendigen Kommunikationsmechanismus auszustatten, auf die von entfernten Prozessen zugegriffen werden soll. Dadurch vermeidet man ein unnötig starkes Größenwachstum der ausführbaren Programme. Allerdings ergeben sich durch diese Optimierung Komplikationen bei der Parameterübergabe im Zusammenhang mit entfernten Methodenaufrufen: Wie sollen Referenzen auf lokale Objekte, d.h. auf Objekte, die nicht für den entfernten Methodenaufruf ausgestattet sind, als Parameter an eine entfernte Methode übergeben werden? In Abschn. 7.3 werden wir sehen, wie

Java diese Problematik angeht.

**Heterogene Systeme.** In der Praxis hat man es meistens mit heterogenen Systemen zu tun; d.h. die Systemkomponenten sind in unterschiedlichen Programmiersprachen geschrieben und werden auf unterschiedlichen Rechnern mit verschiedenen Betriebssystemen und unterschiedlicher Netzwerksoftware ausgeführt. Dies wirft zusätzliche softwaretechnische Probleme auf.

Naturgemäß entstehen diese Probleme vor allem bei der Kommunikation und der Übertragung von Daten und Programmteilen. Beispielsweise können gleich benannte Datentypen auf verschiedenen Rechnern unterschiedlich implementiert oder codiert sein (etwa vier- bzw. zwei-Byte-Integer, low-endian bzw. big-endian-Codierung, unterschiedliche zusammengesetzte Datentypen, wie z.B. Felder). Objekte lassen sich in heterogenen Systemen im Allgemeinen nur äußerst schwierig übertragen, denn die Methoden müssten entweder auch übertragen werden, damit sie auch auf der Empfängerseite zum Aufruf verfügbar wären oder die Methodenimplementierungen müssten auf beiden Seiten vorliegen. Beide Varianten führen bei programmiersprachlicher Heterogenität zu großen Problemen.

Insgesamt bietet aber das objektorientierte Grundmodell mit der klaren Trennung von Schnittstelle und Implementierung auch für die Realisierung heterogener Systeme eine gute Grundlage. Davon ausgehend verbleibt im Wesentlichen die Bewältigung der folgenden Aufgaben:

1. Standardisierung der Basisdatentypen (`int`, `char`, etc.) und ihrer Codierung.
2. Festlegung einer Sprache zur Spezifikation von Schnittstellen.
3. Regeln, die festlegen, welche Bedingungen ein Programm erfüllen muss, damit es als Implementierung einer gegebenen Schnittstelle akzeptiert wird; diese Regeln müssen selbstverständlich für jede Programmiersprache existieren, deren Programme im Rahmen der heterogenen Systeme eingesetzt werden sollen.
4. Mechanismen und Protokolle, mit denen Objekte in verteilten Systemen kommunizieren können.
5. Verzicht auf das Übertragen von Programmcode zwischen verschiedenen Systemteilen.

Die Bewältigung dieser Aufgaben hat sich die Object Management Group, kurz OMG, zum Ziel gesetzt. Die OMG ist von einem Konsortium von mittlerweile hundert von Unternehmen der Informationstechnik und IT-Organisationen (einschließlich aller bedeutenden) ins Leben gerufen worden. Sie beschäftigt sich u.a. mit der Erstellung von Industrienormen und Spezifikationen zur Unterstützung einer allgemeinen Plattform für die Entwicklung

verteilter, heterogener Anwendungen. Ein zentraler Punkt bei diesen Standardisierungsbestrebungen ist der Entwurf und die Spezifikation einer Architektur für die Kommunikation zwischen verteilten Objekten (siehe obigen Punkt 4). Ein Ergebnis dieser Arbeiten ist die **Common Object Request Broker Architecture**, kurz **CORBA** und das CORBA-Komponentenmodell [COR02, Sie00]. Wie im letzten Abschnitt bereits skizziert, ist die Request-Broker-Architektur eine Client-Server-Architektur, bei der der Request-Broker die Client-Anforderungen an geeignete Server vermittelt (siehe Abb. 7.1).

CORBA

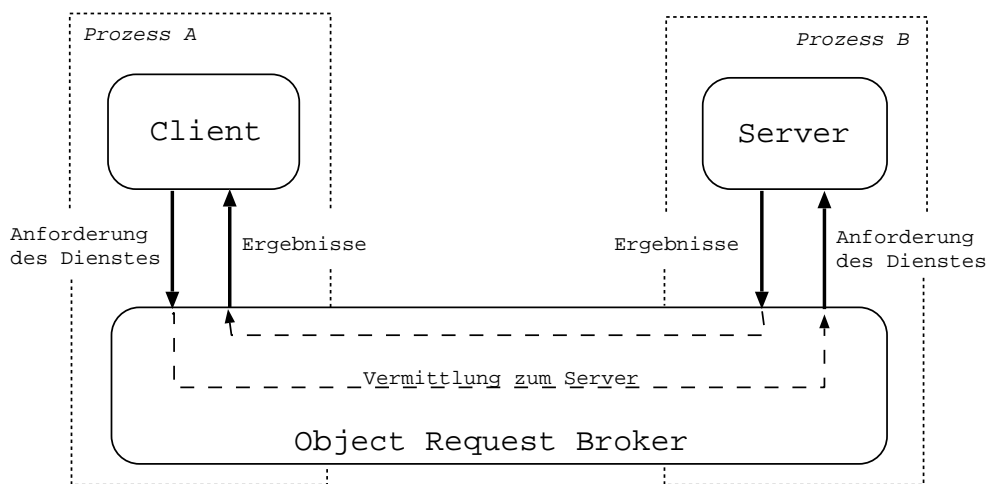


Abbildung 7.1: Request-Broker-Architektur

Die OMG definiert nur die Architektur und die Schnittstellen. Dabei verbleiben noch viele Freiheitsgrade für die Implementierung. Insbesondere ist der Object Request Broker als logische Einheit und nicht als geschlossene Implementierungskomponente zu verstehen. Er kann beispielsweise als Teil der Client- und Server-Prozesse implementiert werden oder auch als jeweils unabhängiger Prozess auf Client- und Server-Seite. Die Realisierung und Vermarktung von CORBA-konformen Softwareprodukten und den darüber hinaus standardisierten Diensten war und bleibt Aufgabe der IT-Industrie. Bekannte CORBA-Implementierungen sind z.B. VisiBroker, Orbus, JacORB und omniORB.

## 7.2 Kommunikation über Sockets

Dieser Abschnitt erläutert die Client-Server-Kommunikation verteilter Prozesse über Sockets. Nach einer kurzen Einführung werden zunächst anhand eines Beispiels die elementaren Schritte zur Realisierung eines einfachen Servers und eines einfachen Clients beschrieben. Danach werden wir wichtige Aspekte im Zusammenhang mit der Internet-Programmierung behandeln und darauf aufbauend den Browser von Kap. 5 internetfähig ma-

chen. Abschließend betrachten wir die Realisierung von Servern mit mehreren Ausführungssträngen.

### 7.2.1 Sockets: Allgemeine Eigenschaften

Daten werden in vielen Netzwerken, vor allem im Internet, in Paketen mit endlicher Länge übertragen. Deshalb ist es im Allgemeinen notwendig, die Daten, die von einem Prozess zu einem anderen geschickt werden sollen, in mehrere Pakete zu zerteilen und am Zielrechner wieder zusammenzusetzen. Dabei fallen eine Reihe nicht trivialer Tätigkeiten an:

- Zerteilen der Daten;
- Einpacken der Daten in Pakete mit Adressinformation, Verwaltungsinformation und Paketinhalt;
- eintreffende Pakete auspacken;
- Reihenfolge der eintreffenden Pakete sicherstellen;
- fehlende Pakete nachfordern;
- Pakete am Zielort wieder zusammensetzen; usw.

Sockets bilden eine Abstraktionsschicht zur Kommunikation zwischen verteilten Prozessen, die den Programmierer von diesen Tätigkeiten befreit. Sockets wurden im Zusammenhang mit Berkeley Unix entwickelt.

Sockets unterstützen die Kommunikation zwischen Clients und Servern. Die Verbindung wird über sogenannte *Ports* an demjenigen Rechner geknüpft, auf dem der Server-Prozess ausgeführt wird. Ports werden über Nummern identifiziert, Rechner über ihre Internetadresse. Das typische Szenario sieht wie folgt aus:

- Der Server-Prozess schließt sich an einen Port seines Rechners an, sagen wir an den Port mit Nummer 8181. Dann wartet er darauf, dass sich ein Client ebenfalls an diesen Port anschließt und damit die Socket-Verbindung zwischen Client und Server herstellt.
- Ein Client, der einen Dienst von einem Server in Anspruch nehmen möchte, muss wissen, auf welchem Rechner der Server-Prozess läuft und an welchem Port der Dienst angeboten wird. Er schließt sich an diesen Port an, kann dann den Dienst abrufen und danach die Socket-Verbindung wieder lösen.

Eine Socket-Verbindung stellt für jede Richtung der Kommunikation zwischen Client und Server einen Strom zur Verfügung, über den Daten ausgetauscht werden können.

Die Stärke der Socket-Kommunikation liegt in der recht einfachen Handhabung. Als Nachteil erweist es sich allerdings in vielen Anwendungen, dass bei der Kommunikation über Sockets und damit über Ströme alle zu übertragenden Daten explizit in Form von Zeichenfolgen zu codieren sind, sodass die Stärken des objektorientierten Programmiermodells teilweise verloren gehen.

Aus objektorientierter Sicht bietet die Socket-Programmierung zwei interessante Aspekte:

1. Von programmtechnischem Interesse ist die Realisierung der Sockets als Objekte.
2. Sie bietet ein Beispiel, um objektorientierte Konzepte auch auf der Architekturebene verstehen zu lernen. Konzeptionell kann es nämlich hilfreich sein, Clients und Server jeweils als einzelne Objekte zu begreifen, auch wenn sie programmtechnisch vielleicht aus vielen Objekten bestehen. Auch auf dieser höheren Abstraktionsebene lassen sich dann objektorientierte Fragestellungen anwenden. Z.B. ist es sinnvoll zu fragen, ob ein Server eine Spezialisierung eines anderen Servers ist, auch wenn die beiden Server mit unterschiedlichen Programmiersprachen implementiert sind.

Im Übrigen werden wir die Kenntnis von Sockets verwenden, um in Abschn. 7.3 bestimmte Aspekte entfernter Methodenaufrufe zu erläutern.

### 7.2.2 Realisierung eines einfachen Servers

In diesem Abschnitt demonstrieren wir die Realisierung eines einfachen Servers und führen damit in die Benutzung des Bibliothekspakets `java.net` ein, das die Klassen für die Socket-Programmierung in Java bereitstellt. Als Beispiel entwickeln wir einen primitiven Datei-Server.

Das grundlegende Muster zur Realisierung eines einfachen Servers sieht wie folgt aus: Der Server-Prozess schließt sich an den gewünschten Port an. Dies geschieht in Java mittels des Konstruktors der Klasse `ServerSocket` (s.u.). Dann betritt er üblicherweise eine Endlosschleife. Jeder Durchlauf durch die Schleife entspricht der Bedienung eines Clients. In der Schleife wartet der Server, dass sich ein Client an den Port anschließt. Das Warten und das nachfolgende Herstellen der Socket-Verbindung zum Client wird in Java von der Methode `accept` geleistet. Ihr Aufruf blockiert, bis sich ein Client angeschlossen hat, und liefert dann das Socket-Objekt zurück, das die Verbindung zum Client repräsentiert. Das Socket-Objekt stellt Methoden zur Verfügung, um die Ein- und Ausgabeströme zum Client zu öffnen. Das folgende Programmfragment fasst dieses Muster zusammen:

```

ServerSocket serversocket = new ServerSocket( Portnummer );
while( true ) {
    Socket socket = serversocket.accept();
    Öffnen der Ein- und Ausgabeströme zum Client
    Kommunikation zwischen Server und Client
    socket.close();
}

```

*Protokoll*

Der zu entwickelnde Datei-Server soll es Clients ermöglichen, Dateien von dem Rechner zu laden, auf dem der Server läuft. Das *Protokoll* dazu ist denkbar einfach: Der Client schickt die Nachricht "GET *Dateiname*" an den Server. Gibt es eine Datei dieses Namens auf dem Server-Rechner (und darf der Server-Prozess auf diese Datei zugreifen), schickt der Server den Datei-Inhalt zum Client. Auf diese Weise können Clients von einem entfernten Rechner eine Datei holen, ohne sich auf diesem Rechner einzuloggen.

**Vorsicht.** Ohne dies hier im Detail thematisieren zu wollen, sei auf die Sicherheitsproblematik hingewiesen, die der Datei-Server ggf. verursacht. Hängt der Server-Rechner am Internet, werden durch den Datei-Server möglicherweise alle Dateien, auf die er zugreifen kann, für die ganze Welt einsehbar.

Abbildung 7.2 zeigt eine mögliche Realisierung des Datei-Servers, die obigem Muster folgt. Der Datei-Server bietet seinen Dienst an Port 8181 an (die Portnummer ist hier beliebig gewählt; vgl. Abschn. 7.2.4 zur Vergabe von Portnummern). Sobald sich ein Client an den Port angeschlossen hat, liefert die `accept`-Methode ein entsprechendes Socket-Objekt. Mit den Methoden `getInputStream` und `getOutputStream` der Klasse `Socket` öffnet der Server den Eingabe- und Ausgabestrom zum Client. Zur einfacheren und effizienteren Handhabung haben wir die vom Socket-Objekt gelieferten Byte-Ströme in gepufferte Zeichenströme eingebettet (vgl. Abschn. 4.3.2.1). Der erste Schritt der Kommunikation zwischen Client und Server besteht darin, dass der Server die Kommandozeile vom Client liest. Die Hilfsmethode `dateiOeffnen` zerteilt die Zeile in das Kommando GET und den Dateinamen und liefert einen `BufferedReader`, der aus der geöffneten Datei liest. Der Einfachheit halber haben wir auf eine Fehlerbehandlung verzichtet (es versteht sich von selbst, dass sich ein derartiges Vorgehen in der Praxis verbietet). Anschließend überträgt der Server die Datei zeichenweise zum Client und weist am Ende den Ausgabestrom an, alle noch gepufferten Zeichen zu übertragen (`zumClient.flush()`). Danach schließt er die Verbindung zum Client.

```

import java.io.*;
import java.net.*;
import java.util.*;

public class DateiServer {
    public static void main(String argv[]) throws IOException {
        ServerSocket serverso = new ServerSocket(8181);
        while( true ) {
            Socket so = serverso.accept();

            // Oeffnen des Eingabestroms vom Client
            BufferedReader vomClient = new BufferedReader(
                new InputStreamReader( so.getInputStream() ) );

            // Oeffnen des Ausgabestroms zum Client
            BufferedWriter zumClient = new BufferedWriter(
                new OutputStreamWriter( so.getOutputStream() ) );

            // Lesen des Kommandos vom Client und Datei-Uebertragung
            try {
                String kommandozeile = vomClient.readLine();
                BufferedReader dateiLeser= dateiOeffnen(kommandozeile);

                int c = dateiLeser.read();
                while( c != -1 ){
                    zumClient.write( c );
                    c = dateiLeser.read();
                }
                dateiLeser.close();
            } catch( Exception e ) {
                new PrintWriter(zumClient). println("Fehler");
            }
            zumClient.flush();
            so.close();
        }
    }

    private static BufferedReader dateiOeffnen( String zeile )
        throws Exception {
        StringTokenizer stok = new StringTokenizer(zeile, " \r\n");
        String kommando = stok.nextToken();
        String dateiName = stok.nextToken();
        if( kommando.equals("GET") ) {
            return new BufferedReader( new FileReader( dateiName ) );
        } else {
            throw new Exception("Syntaxfehler in Kommando");
        }
    }
}

```

Abbildung 7.2: Ein einfacher Datei-Server



### 7.2.3 Realisierung eines einfachen Clients

Dieser Abschnitt erläutert die Realisierung einfacher Clients und führt damit die Einführung in die Benutzung des Bibliothekspakets `java.net` fort. Als Beispiel entwickeln wir einen Client zum Datei-Server des letzten Abschnitts.

Das grundlegende Muster zur Realisierung eines Clients beginnt mit dem Anschluss des Client-Prozesses an den entsprechenden Port des Rechners, auf dem der Server läuft. Dieser Anschluss wird vom Konstruktor der Klasse `Socket` geleistet. Dazu bekommt der Konstruktoraufruf als Parameter die Rechneradresse und die Portnummer mitgegeben. Rechner werden dabei durch ihre Adresse im Internet bezeichnet: entweder als strukturierten Namen der Form `bonsai.fernuni-hagen.de` oder durch die vier Byte lange Internetadresse, z.B. `132.176.114.21`. Das vom Konstruktor erzeugte `Socket`-Objekt repräsentiert die Verbindung zum Server und stellt – wie im letzten Abschnitt erläutert – Methoden zur Verfügung, um die zugehörigen Ein- und Ausgabeströme zu öffnen. Das folgende Programmfragment fasst dieses Muster zusammen:

```
Socket socket = new Socket( Rechneradresse, Portnummer );
```

*Öffnen der Ein- und Ausgabeströme zum Server*

*Kommunikation zwischen Client und Server*

```
socket.close();
```

Zur Illustration dieses Musters wollen wir einen Client entwickeln, der mit dem im letzten Abschnitt beschriebenen Datei-Server zusammenarbeiten kann. Er bekommt beim Start zwei Parameter mitgegeben: die Adresse des Rechners, von dem er die Datei laden soll, und den Dateinamen. Er baut die Verbindung zum Server auf, setzt die Kommandozeile ab, liest dann den Datei-Inhalt aus seinem Eingabestrom und gibt ihn auf der Konsole aus. Abbildung 7.3 zeigt eine mögliche Realisierung für einen solchen Client, die dem beschriebenen Muster entspricht. Programmtechnisch ist dabei zu beachten, dass der Puffer des Ausgabestroms `zumServer`, in den die Kommandozeile geschrieben wurde, mittels der Methode `flush` geleert werden muss<sup>4</sup>, um eine Übertragung an den Server sicherzustellen.

Mit Datei-Server und -Client kann man Dateien von dem Rechner, auf dem der Server läuft, zu dem Rechner übertragen, auf dem der Client läuft. Voraussetzung dafür ist natürlich, dass die Rechner über ein geeignetes Netzwerk verbunden sind. Wir gehen im Folgenden davon aus, dass auf den Rechnern TCP/IP läuft und die Rechner am Internet angeschlossen sind. Für den Fall, dass kein Internet-Anschluss vorhanden ist, kann man die Programme auch lokal ausprobieren, indem man Server und Client auf dem gleichen Rechner ausführt. Dazu ermöglichen es die meisten TCP/IP-Implementierungen, den lokalen Rechner unter dem Namen `localhost` bzw. unter der

<sup>4</sup>Bei Strömen vom Typ `PrintWriter` kann man durch Angabe von `true` als zweitem Konstruktorparameter den `autoflush`-Modus einstellen, der dafür sorgt, dass nach jedem Aufruf von `println` der Puffer des Stroms automatisch geleert wird.

```
import java.io.*;
import java.net.*;

public class DateiClient {
    public static void main(String argv[]) throws IOException {
        String rechneradresse = argv[0];
        String dateiname      = argv[1];

        Socket so = new Socket( rechneradresse, 8181 );

        // Oeffnen des Eingabestroms vom Server
        BufferedReader vomServer = new BufferedReader(
            new InputStreamReader( so.getInputStream() ) );

        // Oeffnen des Ausgabestroms zum Server
        PrintWriter zumServer = new PrintWriter(
            new OutputStreamWriter( so.getOutputStream() ) );

        // Abschicken des Kommandos zum Server
        zumServer. println("GET "+ dateiname );
        zumServer. flush();

        // Lesen und ausgeben der Datei
        int c = vomServer.read();
        while( c != -1 ){
            System.out.print( "" + (char)c );
            c = vomServer.read();
        }
        so.close();
    }
}
```

Abbildung 7.3: Ein einfacher Client zum Datei-Server

Adresse 127.0.0.1 anzusprechen. Zum Testen von Datei-Server und -Client starte man zunächst den Server. Danach kann man den Dienst des Servers bei mehreren Client-Ausführungen nutzen. Schließlich sollte man nicht vergessen, den Server wieder zu beenden (vgl. das in Abschn. 7.2.1 skizzierte typische Anwendungsszenario).

## 7.2.4 Client und Server im Internet

Das obige Beispiel des Datei-Servers demonstriert nicht nur die Anwendung der Socket-Klassen, sondern auch das Prinzip, das den Standarddiensten des Internet zugrunde liegt. Dieser Abschnitt gibt einen kurzen Überblick über einige dieser Dienste und skizziert dann, wie man sich diese Dienste zu Nutze machen kann. Als Beispiel werden wir den in Abschn. 5.3.2 entwickelten Browser internetfähig machen.

Protokoll	Port	Zweck
FTP-DATA	20	Das <b>File-Transfer-Protocol</b> arbeitet an zwei Ports. Über Port 20 werden die Dateien übertragen.
FTP	21	Port 21 wird von FTP für die Übertragung der Kommandos wie „put“ und „get“ genutzt.
SSH	22	<b>Secure Shell</b> ist ein Protokoll für sichere Rechnerverbindungen zum Dialog (Telnet; s.u.) oder Dateitransfer (FTP; s.o.).
Telnet	23	Telnet ist ein Protokoll, um an entfernten Rechnern interaktiv kommandozeilenorientierte Sitzungen durchzuführen.
SMTP	25	Das <b>Simple-Mail-Transfer-Protocol</b> wird für die E-Mail-Übertragung verwendet.
DNS	53	Mit dem <b>Domain-Name-System</b> Protokoll werden z.B. Rechnernamen wie „bonsai.fernuni-hagen.de“ in IP-Adressen übersetzt.
HTTP	80	Das <b>HyperText-Transfer-Protocol</b> ist das zentrale Protokoll des World Wide Web. Eine verschlüsselte Variante ist HTTPS (s.u.).
POP3	110	Das <b>Post-Office-Protocol</b> Version 3 ermöglicht es, auf einem Server angesammelte E-Mail abzuholen.
NNTP	119	Das <b>Network-News-Transfer-Protocol</b> bewältigt die Übertragung der Beiträge zu den News-Gruppen des Usenet.
IRC	194, 6667	Das <b>Internet-Relay-Chat</b> Protokoll wird für Chatdienste verwendet. Eine sichere Variante ist das SILC (s.u.).
HTTPS	443	Das <b>HyperText-Transfer-Protocol Secure</b> ist eine verschlüsselte Variante von HTTP.
SILC	706	Das <b>Secure Internet-Life Conferencing</b> wird für sichere Chatdienste verwendet.
RMI Registry	1099	<b>Remote-Method-Invocation-Registry</b> ist ein Namensdienst für den entfernten Methodenaufruf in verteilten Java-Programmen (vgl. Abschn. 7.3).
SIP	5060	Das <b>Session-Initiation-Protocol</b> ist ein Protokoll zum Aufbau einer Kommunikationssitzung zwischen zwei und mehr Teilnehmern. Das Protokoll wird beim Telefonieren über das Internet eingesetzt.

Abbildung 7.4: Übersicht über eine Auswahl von Internetdiensten

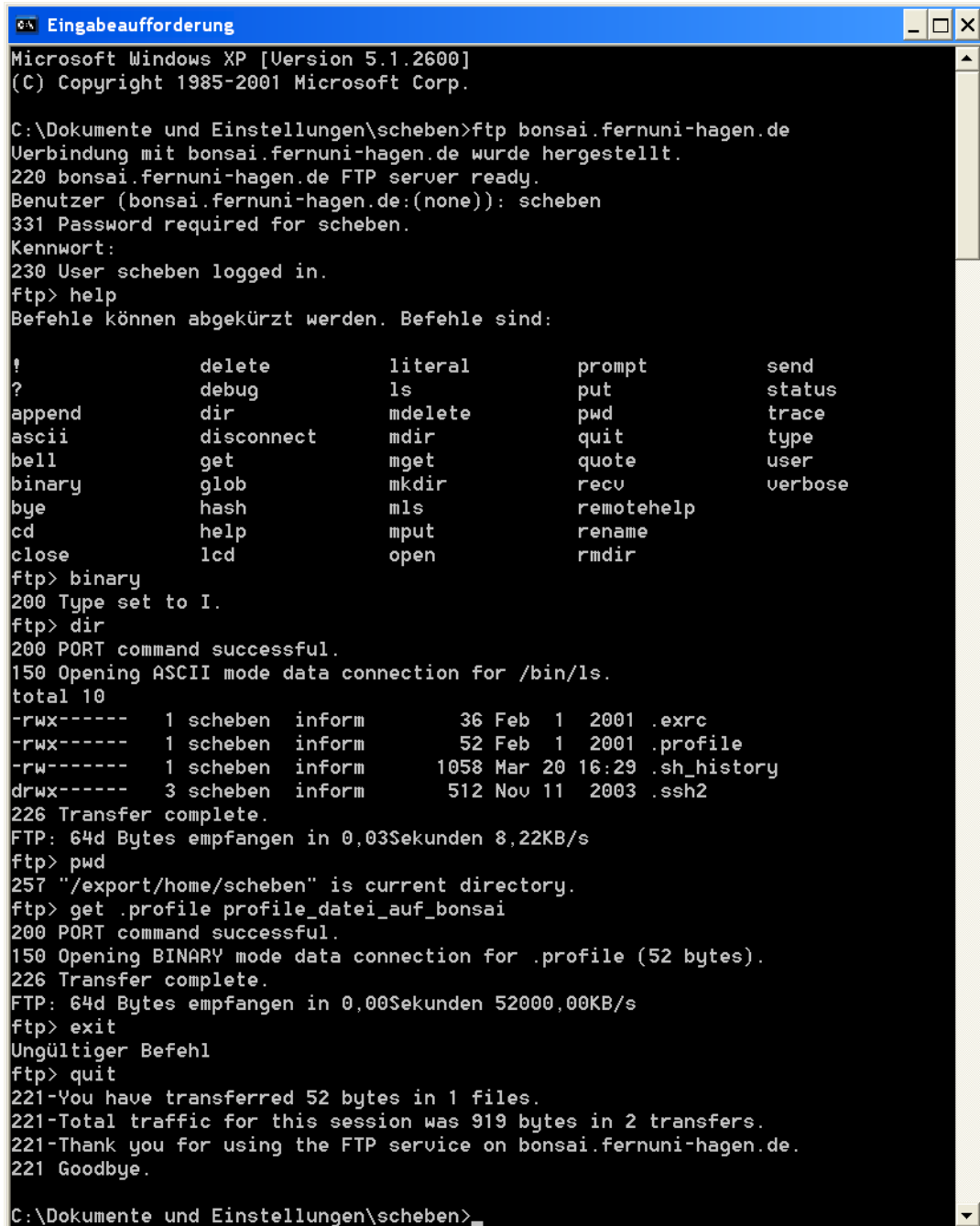
#### 7.2.4.1 Dienste im Internet

Rechner, die am Internet angeschlossen sind, bieten üblicherweise eine Reihe von Standarddiensten an. Damit man diese Dienste von anderen Rechnern aus in Anspruch nehmen kann, gibt es Konventionen darüber, an welchen Ports die Dienste angeboten werden. Die Tabelle in Abb. 7.4 gibt eine kleine Übersicht über verbreitete Internetdienste und ihre Portnummern.

Prinzipiell kommunizieren alle diese Dienste auf die gleiche Weise über Sockets wie der erläuterte Datei-Server. Während der Datei-Server aber nur das Kommando GET versteht und bearbeiten kann, also ein sehr triviales Protokoll besitzt, unterstützen viele der genannten Dienste recht komplexe Protokolle.

**Protokollbeispiel.** Als Beispiel für ein Protokoll wollen wir uns zunächst FTP kurz anschauen. Dazu benutzen wir das Programm `ftp`. In Abbildung 7.5 zeigen wir eine Kommandozeilen-orientierte Kommunikation mit „bonsai.fernuni-hagen.de“. Diese verdeutlicht sehr schön die gegenseitige Kommunikation. Alle Zeilen, die mit dem Prompt „ftp>“ beginnen, enthalten die Kommandos, die an den FTP-Server abgesetzt werden. Die anderen Zeilen zeigen die Antworten des FTP-Servers auf „bonsai.fernuni-hagen.de“.

Zunächst wird mit dem Kommando `ftp bonsai.fernuni-hagen.de` eine FTP-Verbindung zum Rechner aufgebaut. Nach Eingabe des Kommandos verlangt der Rechner zur Authentifizierung Benutzername und Paßwort. Anschließend können FTP-Kommandos eingegeben werden. Zunächst wird der Befehl `help` abgesetzt, über den man eine Liste aller möglichen FTP-Kommandos vom Server abrufen kann. Der Befehl `binary` stellt den Übertragungsmodus für Dateien auf binär. Mit Hilfe des `dir`-Befehls erhält man eine Liste aller Dateien und Unterverzeichnisse zu dem Verzeichnis, in dem man sich auf dem entfernten Rechner befindet. Mit `pwd` kann man das aktuelle Verzeichnis erfragen. Mit `get` kann man eine Datei herunterladen und einen Dateinamen für die Zielfeile auf dem Client-Rechner angeben. Mit `quit` beendet man die aktuelle FTP-Verbindung.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Dokumente und Einstellungen\scheben>ftp bonsai.fernuni-hagen.de
Verbindung mit bonsai.fernuni-hagen.de wurde hergestellt.
220 bonsai.fernuni-hagen.de FTP server ready.
Benutzer (bonsai.fernuni-hagen.de:(none)): scheben
331 Password required for scheben.
Kennwort:
230 User scheben logged in.
ftp> help
Befehle können abgekürzt werden. Befehle sind:

!           delete          literal        prompt        send
?           debug           ls             put            status
append      dir              mdelete       pwd            trace
ascii       disconnect       mdir          quit           type
bell        get               mget          quote          user
binary      glob              mkdir         recu           verbose
bye         hash             mls           remotehelp
cd          help             mput          rename
close      lcd              open          rmdir

ftp> binary
200 Type set to I.
ftp> dir
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 10
-rwx----- 1 scheben inform      36 Feb  1  2001 .exrc
-rwx----- 1 scheben inform      52 Feb  1  2001 .profile
-rw----- 1 scheben inform    1058 Mar 20 16:29 .sh_history
drwx----- 3 scheben inform      512 Nov 11  2003 .ssh2
226 Transfer complete.
FTP: 64d Bytes empfangen in 0,03Sekunden 8,22KB/s
ftp> pwd
257 "/export/home/scheben" is current directory.
ftp> get .profile profile_datei_auf_bonsai
200 PORT command successful.
150 Opening BINARY mode data connection for .profile (52 bytes).
226 Transfer complete.
FTP: 64d Bytes empfangen in 0,00Sekunden 52000,00KB/s
ftp> exit
Ungültiger Befehl
ftp> quit
221-You have transferred 52 bytes in 1 files.
221-Total traffic for this session was 919 bytes in 2 transfers.
221-Thank you for using the FTP service on bonsai.fernuni-hagen.de.
221 Goodbye.

C:\Dokumente und Einstellungen\scheben>
```

Abbildung 7.5: Kommunikation mit einem FTP-Server

Statt der Kommandozeilen-orientierten Kommunikation benutzt man in der Regel geeignete Clients, die die Ordnerstruktur und die enthaltenen Dateien in einer Explorer-ähnlichen Darstellung anzeigen und die tatsächliche Kommunikation verbergen. Abbildung 7.6 zeigt ein solches Tool für Windows. Abbildung 7.7 zeigt, dass man auch den Internet-Explorer für FTP verwenden kann. Die letzte Abbildung zeigt einen SSH-Client, der auch zum Dateitransfer eingesetzt werden kann.

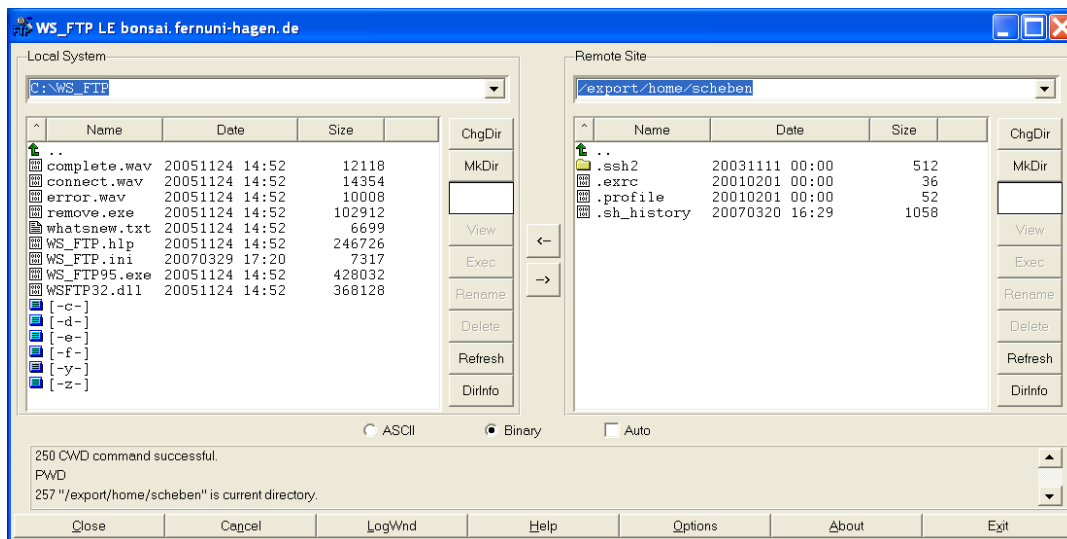


Abbildung 7.6: Kommunikation mit einem FTP-Server über einen FTP-Client unter Windows

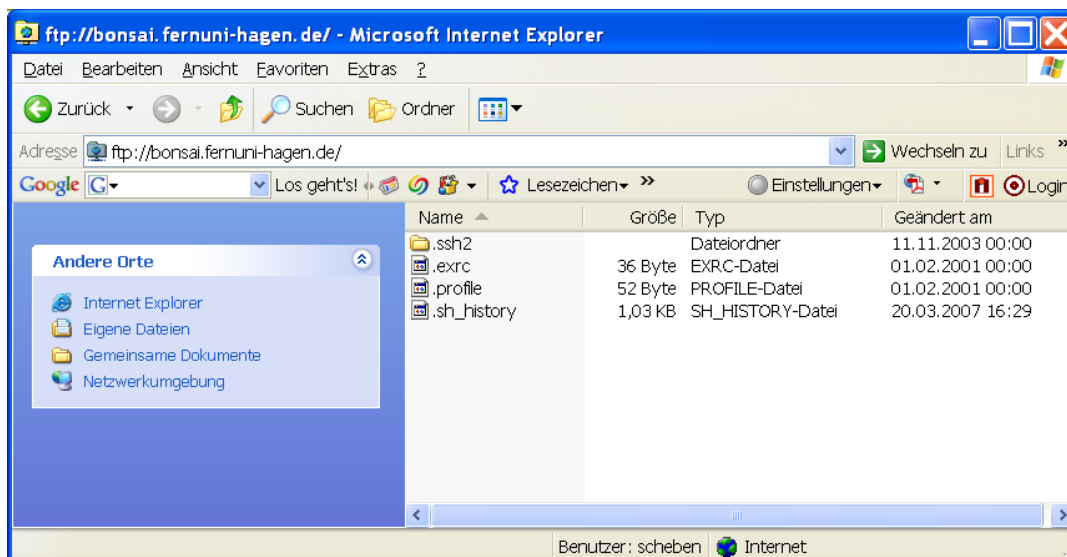


Abbildung 7.7: Kommunikation mit einem FTP-Server mit Hilfe des IE

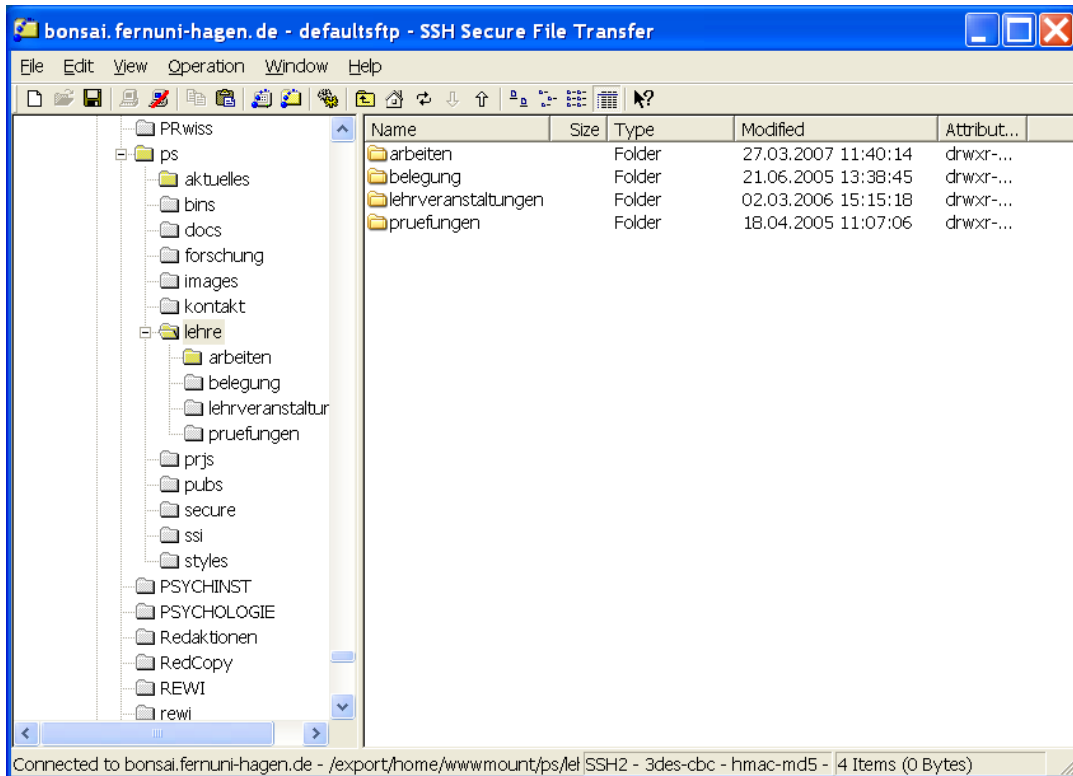


Abbildung 7.8: SSH-Kommunikation zum Dateitransfer

#### 7.2.4.2 Zugriff auf einen HTTP-Server

Um zu verdeutlichen, dass das oben erläuterte Client/Server-Paar zur Dateiübertragung die zentralen programmtechnischen Aspekte der Client/Server-Kommunikation im Internet demonstriert, modifizieren wir den Client so, dass er von einem HTTP-Server lesen kann. Selbstverständlich werden wir nicht das gesamte HTTP-Protokoll unterstützen. Vielmehr soll es uns reichen, dass der Client eine WWW-Seite lesen und ausgeben kann. Abbildung 7.9 zeigt eine mögliche Realisierung, die sich an die Implementierung des Datei-Clients anlehnt (vgl. Abb. 7.3).

Bevor wir die Realisierung kurz besprechen, betrachten wir die Anwendung des HTTP-Clients. Er bekommt als Argument die Adresse der WWW-Seite in Form eines sogenannten **Uniform Resource Locators**, meist kurz als **URL** bezeichnet. Ein URL besteht im Wesentlichen aus drei Teilen: dem verwendeten Protokoll, der Rechneradresse (möglicherweise gefolgt von einer Portnummer) und dem Dateinamen (möglicherweise gefolgt von einer Abschnittsangabe). Beispielsweise beschreibt der URL `http://java.sun.com/index.jsp` eine Datei `index.jsp`, die im Wurzelverzeichnis des HTTP-Servers auf dem Rechner mit Adresse `java.sun.com` liegt. Außer dem HTTP-Protokoll werden von URLs auch `ftp`, `telnet`, andere Protokolle

URL

```

import java.io.*;
import java.net.*;

public class SimpleHTTPClient {

    public static void main(String argv[]) throws IOException {
        URL url = new URL( argv[0] );

        if( url.getProtocol().equals("http") ) {
            Socket so = new Socket( url.getHost(), 80 );

            BufferedReader vomServer = new BufferedReader(
                new InputStreamReader( so.getInputStream() ) );

            PrintWriter zumServer = new PrintWriter(
                so.getOutputStream(), true );

            String getCommand = "GET " + url.getFile() + " HTTP/1.1\r\nHost: "
                + url.getHost()+"\r\n";
            zumServer.println(getCommand);

            // Lesen und ausgeben der WWW-Seite
            int c = vomServer.read();
            while( c != -1 ){
                System.out.print( "" + (char)c );
                c = vomServer.read();
            }
            so.close();
        }
    }
}

```

Abbildung 7.9: Ein einfacher Client zum Laden von WWW-Seiten

und die Variante `file://Dateiname` unterstützt. Am 30.3.07 wurde der einfache HTTP-Client von Abb. 7.9 mit obigem URL aufgerufen. Abbildung 7.10 zeigt die Antwort des HTTP-Servers: Er liefert zunächst allgemeine Protokollinformationen und dann den Inhalt der Datei (beginnend mit `<!DOCTYPE HTML ...`).

In der Realisierung des HTTP-Clients wird die Klasse `URL` aus `java.net` benutzt, die einen bequemen Umgang mit URLs ermöglicht. Der Konstruktoraufbau prüft, ob der übergebene String einen syntaktisch korrekten URL bezeichnet. Mittels der Methoden `getProtocol`, `getHost` und `getFile` kann man sich die Teile eines URL-Objekts geben lassen. Der Strom zum Server wird als `PrintWriter` mit automatischem Flush implementiert (vgl. die entsprechende Fußnote auf S. 446). Das GET-Kommando aus dem HTTP-Protokoll bekommt außer dem Dateinamen auch die HTTP-Version und ggf. den Host als Parameter. Ansonsten unterscheidet sich der HTTP-Client nicht vom Datei-Client.



```

HTTP/1.1 200 OK
Server: Sun-ONE-Web-Server/6.1
Date: Fri, 30 Mar 2007 13:23:02 GMT
Content-type: text/html; charset=ISO-8859-1
Set-cookie: JSESSIONID=2E4F358DC96208BF484249068A5DCA9A; Path=/
Transfer-encoding: chunked
Connection: close

2000

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                        "http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
<title>Java Technology</title>
<meta name="keywords" content="Java, platform">
<meta name="collection" content="reference">
<meta name="description" content="Java technology is a portfolio
                                of products that are based on the power of networks
                                and the idea that the same software should run on many
                                different kinds of systems and devices.">
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<meta name="date" content="2007-03-29">
<link href="http://developers.sun.com/rss/java.xml" rel="alternate"
      type="application/rss+xml" title="rss" />
</head>
<!--stopindex-->

<link rel="stylesheet" href="/css/default_developer.css" />

<!-- END METADATA -->
<script type="text/javascript" language="JavaScript"
      src="/js/popUp.js"></script>
<script language="javascript1.2" type="text/javascript"
      src="/js/sniff.js"></script>
...
</head>
<!--stopindex-->

<body leftmargin="0" topmargin="0" marginheight="0" marginwidth="0"
      rightmargin="0" bgcolor="#ffffff" onload="prepmenus();prephome();
      done = true" class="a0v0">

<script language="JavaScript">
<!--
var s_pageName="home page"
//--></script>

...

</body>
</html>

```

Abbildung 7.10: Antwort eines HTTP-Servers

### 7.2.4.3 Netzsurfer im Internet

Die oben skizzierte Technik können wir uns zu Nutze machen, um den Browser von Kap. 5 so zu erweitern, dass er WWW-Seiten aus dem Internet laden kann. Im Zuge dessen werden wir auch eine etwas komfortablere Technik zeigen, auf URLs zuzugreifen.

Die Browser vom Typ `NetzSurfer` (vgl. Abb. 5.22) können WWW-Seiten vom lokalen Dateisystem laden und anzeigen. Dabei wird ein sehr stark eingeschränkter HTML-Sprachschatz unterstützt. Seiten, die diesem Sprachschatz genügen, werden entsprechend formatiert, andere Seiten werden unformatiert ausgegeben. Die Spracheinschränkung bzgl. HTML aufzuheben ist nicht Gegenstand dieser Einführung in objektorientierte Konzepte. Von programmtechnischem Interesse ist es allerdings zu demonstrieren, wie elegant man durch Abstraktion den Zugriff auf Informationen über unterschiedliche Protokolle vereinheitlichen kann. (Die Abstraktion wird im Folgenden demonstriert an Hand der Methode `laden` der Klasse `NetzSurfer2` aus Abbildung 7.12, die aufgrund der Verwendung der Klasse `java.net.URL` nicht mehr zwischen den verschiedenen Protokollen unterscheiden muss.)

Wir statten also die Browser mit der Fähigkeit aus, die WWW-Seite zu einem gegebenen URL zu laden. Abbildung 7.11 zeigt drei Anwendungsbeispiele des erweiterten Browsers:

- Im ersten Beispiel wurde eine MHTML-Seite vom lokalen Dateisystem geladen; dazu musste dem Dateinamen der Protokollbezeichner `file` vorangestellt werden.
- Im zweiten Beispiel wurde die gleiche Seite von einem HTTP-Server geladen.
- Im dritten Beispiel wurde die oben schon erwähnte Seite mit URL `http://java.sun.com/index.jsp` geladen. Da sie die HTML-Einschränkungen des Browsers nicht befolgt, erscheint sie unformatiert.

Für die Erweiterung des Browsers brauchen wir nur den Ladevorgang zu verallgemeinern (auf eine angemessene Fehlerbehandlung verzichten wir hier). Dazu realisieren wir eine Klasse `NetzSurfer2`, in der wir die Methode `laden` der Klasse `NetzSurfer` aus Abb. 5.22, S. 357, überschreiben. In der überschreibenden Methode könnten wir die Seitenadresse, die der Methode `laden` als String übergeben wird, zur Erzeugung eines URL-Objekts heranziehen und dann eine Fallunterscheidung über die unterstützten Protokolle machen, in der jedes Protokoll entsprechend behandelt wird.

Diese Vorgehensweise ist unnötig kompliziert. Insbesondere müssten wir dann die jeweils unterschiedliche Verwaltungsinformation des verwendeten Protokolls von dem Inhalt der übertragenen Seite trennen. Dies kann man beim Vergleich von Abb. 7.10 mit dem dritten Fenster in Abb. 7.11 erkennen.

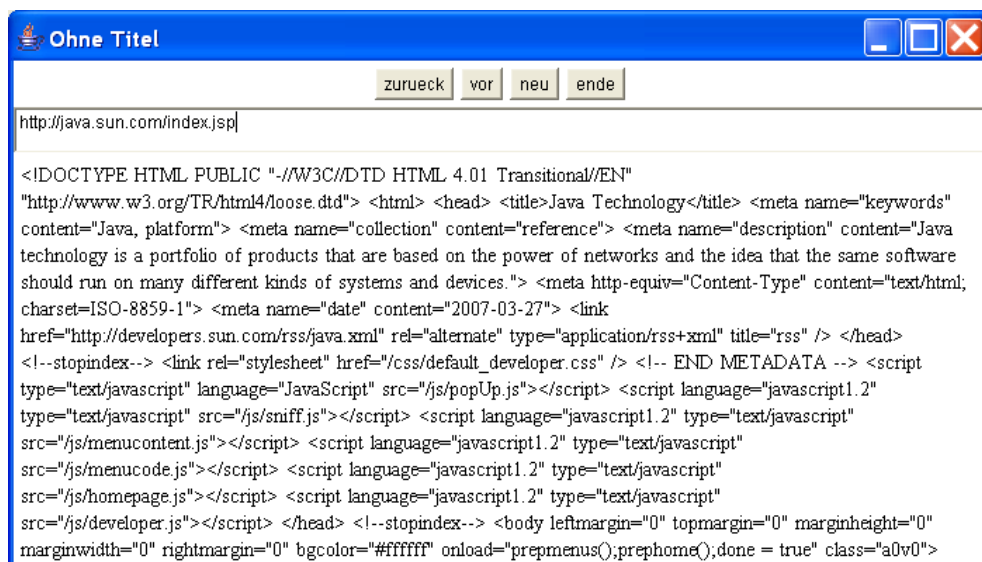
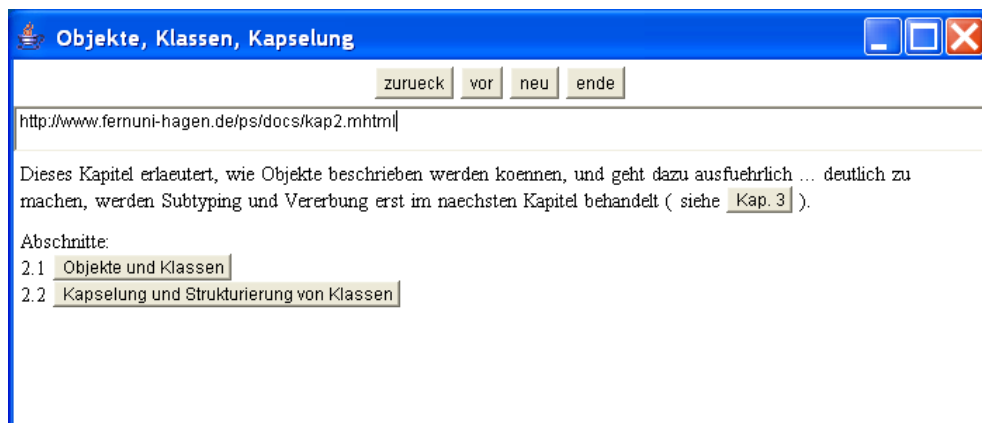
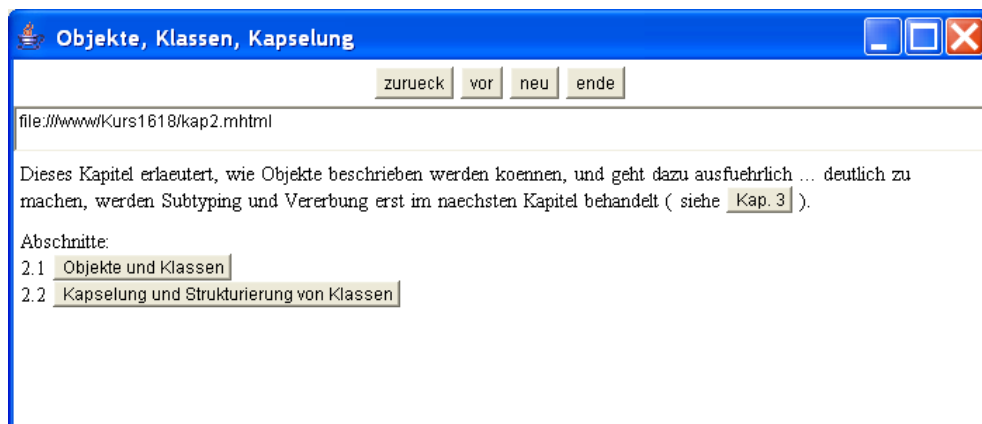


Abbildung 7.11: Browser mit URL-Zugriff

Wie Abb. 7.10 zeigt, liefert das GET-Kommando zunächst einige Zeilen Verwaltungsinformation und erst dann den Seiteninhalt. Im Browser möchten wir aber nur den Inhalt anzeigen. Glücklicherweise bietet die Klasse `URL` auch eine Methode `getContent`, die direkt einen Eingabestrom liefert, aus dem man den Inhalt der vom `URL` bezeichneten Quelle lesen kann. Die in Abb. 7.12 gezeigte Realisierung macht sich dies zu Nutze; insbesondere wird eine spezielle Behandlung unterschiedlicher Protokolle damit überflüssig.

```
import java.io.*;
import java.net.*;

class NetzSurfer2 extends NetzSurfer {

    public void laden( String saddr ) {
        try {
            String titel, inhalt, s;

            URL url = new URL( saddr );
            InputStream is = (InputStream) url.getContent();
            BufferedReader vomServer = new BufferedReader(
                new InputStreamReader( is ) );

            StringWriter sw = new StringWriter();
            int c = vomServer.read();
            while( c != -1 ){
                sw.write( c );
                c = vomServer.read();
            }
            s = sw.toString();
            ... // Fortsetzung wie in Klasse NetzSurfer
        }
    }
}
```

Abbildung 7.12: Überschreiben der Methode `laden`

Bei der Erweiterung des Browsers ist im Übrigen darauf zu achten, dass der Konstruktoraufruf zum Starten eines neuen Browsers in der Klasse `BedienFenster` durch einen Aufruf von `NetzSurfer2` zu ersetzen ist.

### 7.2.5 Server mit mehreren Ausführungssträngen

In Abschn. 7.2.2 haben wir die Realisierung von einfachen Servern behandelt. Server, die nach dem dort vorgestellten Muster arbeiten, können immer nur einen Client zur Zeit bedienen, da sie während der Bedienung keine weiteren Clients akzeptieren können. Diese Einschränkung ist in vielen Fällen inakzeptabel, z.B. wenn viele Clients nahezu gleichzeitig Anfragen stellen, wenn

die Bedienung eines Clients lange Zeit in Anspruch nimmt oder wenn die Verbindung zwischen Server und Client für eine längere Dauer angelegt ist. Das Problem löst man üblicherweise dadurch, dass man den Haupt-Thread des Servers nur die Verbindung zu Clients herstellen lässt. Die Bedienung der Clients wird dann anderen Threads übertragen. Dabei ist es am einfachsten, für jeden Client einen neuen Thread zu starten. Insgesamt ergibt sich daraus folgendes Realisierungsmuster:

```
ServerSocket serversocket = new ServerSocket( Portnummer );
while( true ) {
    Socket socket = serversocket.accept();
    Thread bediener = new ServiceThread( socket );
    bediener.start();
}
```

Die Klasse `ServiceThread` könnte zum Beispiel wie folgt realisiert werden:

```
class ServiceThread extends Thread {
    private Socket so;

    public ServiceThread( Socket cs ) {
        so = cs;
    }

    public void run() {
        Öffnen der Ein- und Ausgabeströme zum Client
        Kommunikation zwischen Server und Client
        so.close();
    }
}
```

Die Anwendung dieses Realisierungsmusters demonstrieren wir im folgenden Abschnitt.

### 7.3 Kommunikation verteilter Objekte über entfernten Methodenaufruf

Dieser Abschnitt erläutert die Kommunikation von Objekten in unterschiedlichen Prozessen über Methodenaufruf. Man spricht häufig auch von der Kommunikation *verteilter* Objekte über *entfernten* Methodenaufruf (engl. **Remote Method Invocation**) und meint damit, dass die Objekte, die ein softwaretechnisches System realisieren, auf mehrere Prozesse und ggf. mehrere Rechner verteilt sind und dass ein Objekt die Methoden entfernter Objekte aufrufen kann.

Der Abschnitt behandelt zunächst die Problematik entfernter Methodenaufrufe und erläutert dann die programmtechnischen Aspekte der Realisierung in Java.

### 7.3.1 Problematik entfernter Methodenaufrufe

In die konzeptionellen Aspekte der Kommunikation über entfernten Methodenaufruf wurde bereits in Abschn. 7.1.2 kurz eingeführt. Diese Aspekte sollen hier vertieft werden. Zur Illustration werden wir Sockets verwenden, um entfernte Methodenaufrufe zu simulieren, und abschließend diese Lösung kurz diskutieren.

#### 7.3.1.1 Behandlung verteilter Objekte

Bei einem verteilten objektorientierten System gehören die Objekte im Allgemeinen zu verschiedenen Prozessen. Als Beispiel betrachten wir die Kom-

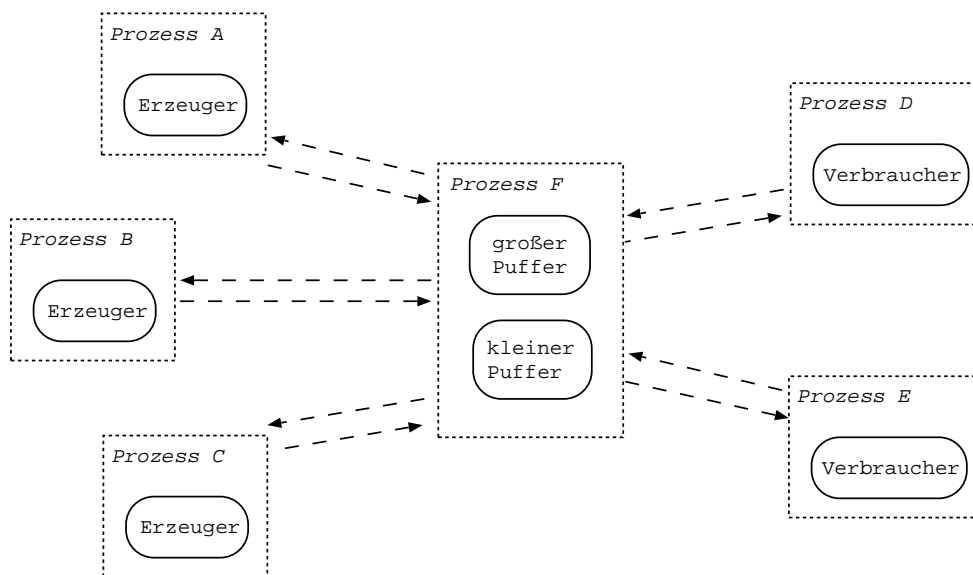


Abbildung 7.13: Erzeuger/Verbraucher-Szenario

munikation von mehreren Erzeugern und Verbrauchern über entfernte Puffer (vgl. die lokale Realisierung des Erzeuger-Verbraucher-Problems aus Unterabschn. 6.2.2.3, S. 410 ff). Abbildung 7.13 zeigt ein Szenario mit einem Server-Prozess, der einen großen und einen kleinen Puffer verwaltet, sowie mit drei Erzeuger- und zwei Verbraucher-Prozessen, die auf die Puffer des (entfernten) Servers zugreifen, um ihre Produkte abzulegen bzw. zu holen.

Durch die Verteilung der Objekte eines Systems auf mehrere Prozesse ergeben sich drei zusätzliche Fragestellungen:

- Unter welcher Bezeichnung kann man entfernte Objekte ansprechen?

- Wie sollen Parameter- und Ergebnis-Objekte bei einem entfernten Methodenaufruf behandelt werden?
- Aus welchen Programmteilen setzen sich die Prozesse zusammen bzw. müssen alle Programmteile in jedem Prozess enthalten sein, damit die Prozesse kommunizieren können?

Im Folgenden werden wir diese drei Fragestellungen kurz erläutern.

**Bezeichnung entfernter Objekte.** Da Objektreferenzen nur relativ zu einem Prozess gültig sind, benötigt ein Prozess, der eine entfernte Methode aufrufen möchte (im Beispiel also ein Erzeuger oder Verbraucher), einen speziellen Mechanismus, mit dem er entfernte Objekte bezeichnen kann. Sinnvollerweise verwendet man dazu ein Adressierungsschema, das einem URL entspricht und aus der Rechneradresse und Portnummer des Servers und einem Objektnamen besteht, mit dem man unterschiedliche Objekte auf Server-Seite unterscheiden kann. Beispielsweise könnten wir den Objekten im Prozess F von Abb. 7.13 die Namen `grosserPuffer` und `kleinerPuffer` geben. Der Server hat dann u.a. die Aufgabe dem Objektnamen die Referenz auf das tatsächliche Objekt zuzuordnen.

**Behandlung von Parametern und Ergebnissen.** Werte lassen sich in der verteilten Programmierung genauso wie in der lokalen Programmierung behandeln, da sie keinen Zustand und keinen Aufenthaltsort (beschrieben durch eine Adresse) haben (vgl. Unterabschn. 1.3.1.1). Die zu einem verteilten System gehörenden Prozesse müssen sich nur über die Repräsentation der Werte verständigen, beispielsweise wie `int`-Werte zu codieren sind. Im Übrigen lassen sich Werte, also insbesondere alle Werte der Basisdatentypen, problemlos zwischen Prozessen austauschen. Bei Objekten bzw. Objektreferenzen ist der Sachverhalt komplexer. Betrachten wir die Methode `ablegen`, die als Parameter ein Objekt vom Typ `Produkt` hat. Prinzipiell gibt es drei Möglichkeiten, die Parameterübergabe zu realisieren:

1. Das Produkt-Objekt bleibt beim Erzeuger; dem Pufferserver wird nur eine Referenz auf das Produkt-Objekt übergeben.
2. Das Produkt-Objekt wird zum Pufferserver „verschoben“ und dementsprechend beim Erzeuger gelöscht.
3. Von dem Produkt-Objekt wird beim Pufferserver eine Kopie erzeugt. Es existieren dann also zwei Versionen des Objekts.

Alle drei Möglichkeiten bergen Vor- und Nachteile.

- Bei der ersten Lösungsvariante entstehen viele Referenzen auf entfernte Objekte. Dies bringt gravierende Effizienzprobleme mit sich: Jeder Zugriff auf ein Parameter-Objekt muss übers Netz abgewickelt werden. Jedes Objekt, das als Parameter übergeben werden soll, muss mit den Kommunikationsmechanismen ausgestattet werden, die für den Umgang mit entfernten Objekten nötig sind.
- Bei der zweiten Lösungsvariante stellt sich die Frage, wie lokale Referenzen auf das zu löschende Objekt zu behandeln sind. Zum einen ist es nicht ohne weiteres möglich, alle Referenzen auf ein Objekt zu ermitteln, und zum anderen führt diese Lösung zu ähnlichen Ineffizienzen wie die erste Variante, da die ehemals lokalen Referenzen jetzt auf das verschobene und dadurch nun auf einem entfernten Rechner liegende Objekt zeigen müssen.
- Die dritte Lösungsmöglichkeit vermeidet das Entstehen einer großen Zahl von Referenzen auf entfernte Objekte. Dieser Vorteil wird aber durch zwei Nachteile erkauft:
  - Erstens reicht es im Allgemeinen nicht, nur die Parameter-Objekte zu kopieren, sondern es müssen auch alle Objekte kopiert werden, die von den Parameter-Objekten aus erreichbar sind. (Andernfalls müssten die Attribute der kopierten Objekte Referenzen auf entfernte Objekte enthalten – mit den oben erläuterten Nachteilen.)
  - Zweitens geht die Identität der Parameter-Objekte durch das Kopieren verloren: Auf der Server-Seite entsteht ein neues Objekt (vgl. Unterabschn. 4.3.2.2). Wird der Zustand dieses Objekts verändert, hat dies keine Auswirkungen auf den Zustand des Objekts, das beim Aufruf als Parameter angegeben wurde. D.h. diese Variante führt zu einer Semantik für den entfernten Methodenaufruf, die sich von der Semantik des lokalen Methodenaufrufs unterscheidet. Der Programmierer muss diesen semantischen Unterschied lernen und geeignet berücksichtigen.

Für den entfernten Methodenaufruf von Java wird eine Kombination aus der ersten und dritten Lösungsmöglichkeit verwendet. Die wichtigsten Aspekte dieser Realisierung werden wir in Unterabschn. 7.3.2.3 erläutern.

**Verteilung von Programmcode** Damit Objekte ihre Methoden ausführen können, muss der entsprechende Programmcode verfügbar sein (in Java sind das die entsprechenden .class-Dateien). Befinden sich die Objekte alle in einem Prozess, muss dieser Prozess Zugriff auf den Programmcode für alle Objekte haben. Verteilen sich die Objekte auf mehrere Prozesse, reicht es, dass



jeder Prozess über den Programmcode derjenigen Objekte verfügt, auf die während der Ausführung in dem Prozess zugegriffen wird; d.h. insbesondere, dass nicht jeder Prozess über den gesamten Programmcode des verteilten Systems verfügen muss.

Geht man davon aus, dass in dem betrachteten verteilten System Objekte von einem Prozess zu einem anderen verschoben werden oder dass von ihnen Kopien bei einem entfernten Prozess angelegt werden (obige Lösungsvarianten 2 und 3), dann kann im Allgemeinen weder zur Übersetzungszeit noch beim Starten eines Prozesses festgestellt werden, welchen Programmcode ein Prozess benötigt. Dazu betrachten wir folgendes Beispiel.

Client-Prozess	Server-Prozess
<pre> class PT {     // ...     public void machwas() {         // Implementierung PT     }     // ... }  class PS extends PT {     // ...     public void machwas() {         // Implementierung PS     }     // ... }  interface ServerInterface ... {     void m (PT param); }  class Client {     public static void main (String[] args) {         // Referenz auf Server beschaffen         ServerInterface server = ...          // Methode m auf Server aufrufen         server.m (new PS());     } } </pre>	<pre> class PT{     // ...     public void machwas() {         // Implementierung PT     }     // ... }  interface ServerInterface {     void m (PT param); }  public class Server ... implements     ServerInterface {     // ...     public void m (PT param) {         param.machwas();     } } </pre>

Abbildung 7.14: Verteilung von Klassen und Schnittstellen auf Prozesse

Ein Server-Prozess implementiert eine Schnittstelle `ServerInterface`, die eine Methode `m` mit einem Parameter vom Typ `PT` bereitstellt. Der Typ `PT` besitzt eine Methode `machwas`, die vom Server auf dem Parameterobjekt aufgerufen wird. Auf Server-Seite sind der Schnittstellentyp `ServerInterface` und der Parametertyp `PT` bekannt.

Ein entfernter Client arbeitet mit Objekten vom Typ `PS`, wobei `PS` ein Subtyp von `PT` ist, in dem die Methode `machwas` von `PT` überschrieben ist (siehe Abbildung 7.14). Auf der Clientseite sind der Schnittstellentyp `ServerInterface` und die Typen `PT` sowie `PS` bekannt. Ruft der Client auf dem Server die Methode `m` mit einem Objekt vom Typ `PS` auf, steht der Server vor der Aufgabe, auf dem übergebenen Objekt die Methode `machwas` auszuführen. Dazu benötigt der Server den Programmcode der Klasse `PS`, über den er aber im Allgemeinen, wie auch in unserem Beispiel, nicht verfügt (beispielsweise könnte es sein, dass der Client und insbesondere die Klasse `PS` erst nach dem Start des Servers implementiert wurden).

Im Folgenden werden wir die Fragestellung der Verteilung von Programmcode nur insoweit behandeln, als es für die Realisierung einfacher verteilter Systeme mit entferntem Methodenaufruf notwendig ist. Es soll aber an dieser Stelle nicht unerwähnt bleiben, dass es für dieses Problem unterschiedliche Lösungsansätze gibt (siehe z.B. [JLHB88]). Java bietet dafür Mechanismen zum dynamischen Laden der Klassen entfernter Objekte. Der obige Server kann in Java beispielsweise so ausgelegt werden, dass er im skizzierten Szenario den Programmcode der Klasse `PS` dynamisch vom Client oder einer dritten Stelle nachlädt.

#### 7.3.1.2 Simulation entfernter Methodenaufrufe über Sockets

Entfernte Methodenaufrufe lassen sich wie folgt über Sockets simulieren. Das entfernte Objekt wird Teil eines Server-Prozesses. Der Server bietet seinen Dienst an einem bestimmten Port an, empfängt vom Client den Objektnamen und den Namen der aufzurufenden Methode. Über den Objektnamen ermittelt er das Objekt, für das die Methode aufgerufen werden soll. Besitzt die Methode Parameter, liest er diese vom Client. Parameter-Objekte werden dabei zusammen mit den von ihnen aus erreichbaren Objektgeflechten serialisiert und beim Server in Kopie angelegt (Lösungsvariante 3, siehe letzten Absatz). Der Server ruft die Methode lokal auf. Nach Beendigung des Aufrufs muss das Ergebnis zum Client übertragen werden (ggf. wieder mittels Objektserialisierung). Entsprechend müsste bei einer abrupten Terminierung das Ausnahmeobjekt an den Client weitergeleitet werden. Besitzt die Methode kein Ergebnis, muss dem Client in geeigneter Weise mitgeteilt werden, dass der Aufruf beendet ist. Wir übertragen dazu das Wort „return“.

Zur Demonstration des skizzierten Vorgehens betrachten wir eine verteilte Realisierung des Erzeuger-Verbraucher-Beispiels. (Auf eine angemessene Ausnahmebehandlung verzichten wir dabei.) Da die Erzeuger und Verbraucher die Methoden des Ringpuffers aufrufen, muss der Pufferprozess als Server ausgelegt werden und ein Protokoll anbieten, das die Funktionalität der Methoden `ablegen` und `holen` simuliert. Die Erzeuger und Verbraucher treten als Clients auf. Erzeuger, Verbraucher und Pufferserver werden durch

drei unabhängige Programme implementiert. Alle drei Programme müssen allerdings die Klasse `Produkt` kennen (vgl. Abb. 6.9). Da Produkt-Objekte als Parameter übergeben werden sollen, muss die Klasse `Produkt` die Schnittstelle `Serializable` implementieren (vgl. Unterabschn. 4.3.2.2, Seite 278).

**Realisierung der Server-Seite.** Damit der Pufferserver mehrere Aufrufe parallel verarbeiten kann, realisieren wir ihn nach dem Muster von Abschn. 7.2.5. Der Server erzeugt die beiden von ihm verwalteten `RingPuffer`-Objekte und wartet auf Anfragen von Clients, die er dann von gesonderten Threads bedienen lässt:

```
public class RingPufferServer {
    static RingPuffer grPuffer;
    static RingPuffer klPuffer;

    public static void main(String[] args) throws Exception {
        grPuffer = new RingPuffer( 50 );
        klPuffer = new RingPuffer( 4 );

        ServerSocket serversocket = new ServerSocket(1199);
        while( true ) {
            Socket socket = serversocket.accept();
            Thread bediener = new ServiceThread( socket );
            bediener.start();
        }
    }
}
```

Abbildung 7.15 zeigt die Realisierung der Service-Threads. Die `run`-Methode des Service-Threads besteht im Wesentlichen aus drei Teilen:

1. Lesen des Objektnamens und des Kommandos. Da über den Strom von den Clients ggf. Objekte zu übertragen sind, wird er als Objektstrom realisiert.
2. Ermitteln des Objekts, für das die Methode aufgerufen werden soll.
3. Übertragen der Parameter, Aufruf der Methode und Rückgabe der Ergebnisse bzw. der Terminierungsantwort „return“.

Die Klasse `RingPuffer` kann unverändert von der nicht-verteilten Realisierung übernommen werden (vgl. Abb. 6.10, S. 413).

```

class ServiceThread extends Thread {
    private Socket so;
    public ServiceThread( Socket cs ) { so = cs; }

    public void run() {
        try {
            // Lesen des Objektnames und des Kommandos
            ObjectInputStream vomClient =
                new ObjectInputStream( so.getInputStream() );
            StringWriter sw = new StringWriter();
            int c = vomClient.readByte();
            while( c != '\n' ) {
                sw.write( c );
                c = vomClient.readByte();
            }
            String zeile = sw.toString();
            StringTokenizer stok = new StringTokenizer( zeile, " ");
            String objName = stok.nextToken();
            String kommando = stok.nextToken();
            RingPuffer lager;

            // Feststellen der Objektreferenz zum Objektnamen
            if( objName.equals("grosserPuffer") ) {
                lager = RingPufferServer.grPuffer;
            } else if( objName.equals("kleinerPuffer") ) {
                lager = RingPufferServer.klPuffer;
            } else throw new Exception("Objektnamen unbekannt");

            // Aufruf der entsprechenden Methode
            if( kommando.equals("ablegen") ) {
                // Abwickeln des Methodenaufrufs
                Produkt prd = (Produkt) vomClient.readObject();
                lager.ablegen( prd );
                PrintWriter zumClient = new PrintWriter(
                    so.getOutputStream(), true );
                zumClient.println("return");
            } else if( kommando.equals("holen") ) {
                // Abwickeln des Methodenaufrufs
                Produkt prd = lager.holen();
                ObjectOutputStream zumClient =
                    new ObjectOutputStream( so.getOutputStream() );
                zumClient.writeObject( prd );
                zumClient.flush();
            } else throw new Exception("Syntaxfehler in Kommando");
            so.close();
        } catch( Exception e ) {
            System.out.println("Fehler: " + e ); System.exit( -1 );
        }
    }
}

```

Abbildung 7.15: Die Klasse ServiceThread

**Realisierung der Client-Seite.** Am Beispiel der Realisierung von Erzeugern betrachten wir die Aspekte der Client-Seite. Ausgangspunkt ist die Klasse `Erzeuger` von S. 410. An dieser Klasse müssen wir im Wesentlichen drei Änderungen vornehmen.

1. Die Klasse bekommt eine `main`-Methode, die drei Parameter erwartet: die Adresse des Rechners, auf dem der zu benutzende Pufferserver läuft; die Portnummer des Servers; den Namen des Puffers, in den die Produkte abgelegt werden sollen.
2. Der Konstruktor bekommt nicht mehr die Referenz auf das `RingPuffer`-Objekt als Parameter, sondern dessen Beschreibung in Form der Rechneradresse, der Portnummer und des Objektnamens.
3. Der Aufruf `lager.ablegen(prd)` wird ersetzt durch das Herstellen einer Socket-Verbindung zum Server und die Übertragung des Objektnamens, des Methodennamens und des Produkts. Der Erzeuger darf erst dann weiter arbeiten, wenn ihn der Pufferserver darüber benachrichtigt hat, dass der Aufruf der Methode `ablegen` auf Server-Seite terminiert hat.

Abbildung 7.16 zeigt eine entsprechende Realisierung.

**Diskussion.** Die Simulation entfernter Methodenaufrufe über Sockets diente im Wesentlichen dazu, die zentralen Aspekte des entfernten Methodenaufrufs auf der Basis bekannter programmtechnischer Konstrukte vorzustellen und deren direkte Unterstützung durch das Programmiersystem zu motivieren und vorzubereiten. Für die praktische Anwendung bringt die Socketrealisierung entfernter Methodenaufrufe im Vergleich zu einer direkten Unterstützung erhebliche Nachteile mit sich: Die Parameterübergabe und Fehlerbehandlung muss für jeden entfernten Methodenaufruf ausprogrammiert werden. Dies bläht den Programmtext auf, erhöht die Fehleranfälligkeit und führt dazu, dass Programme zum Teil erheblich geändert werden müssen, um sie im Rahmen einer verteilten Architektur zu nutzen. Beispielsweise hat die `run`-Methode der Klasse `Erzeuger` ihr Erscheinungsbild deutlich verändert; wie wir im nächsten Abschnitt sehen werden, sind derartige Änderungen unnötig, wenn man einen Mechanismus für entfernte Methodenaufrufe nutzt, der vom Programmiersystem direkt unterstützt wird. Eine direkte Unterstützung bringt darüber hinaus die üblichen Vorteile der automatischen Typprüfung und Ausnahmebehandlung mit sich.

### 7.3. KOMMUNIKATION ÜBER ENTFERNTEN METHODENAUFBRUF 467

```
public class Erzeuger extends Thread {
    public static void main( String[] args ) throws Exception {
        String rAdr = args[0];
        int    pNr  = Integer.parseInt( args[1] );
        String oId  = args[2];
        new Erzeuger( rAdr, pNr, oId ). start();
    }

    // anstelle von private RingPuffer lager;
    private String rechnerAdr;
    private int    portNr;
    private String objName;

    Erzeuger( String ra, int pn, String onm ) {
        rechnerAdr = ra;
        portNr      = pn;
        objName      = onm;
    }

    public void run(){
        while( true ){
            Produkt prd = new Produkt();
            try{
                sleep( prd.getProduktionsZeit() ); //"Produktionszeit"

                // anstelle von lager.ablegen( prd );
                Socket so = new Socket( rechnerAdr, portNr );

                // Abschicken des Kommandos und Produkts zum Server
                ObjectOutputStream zumServer =
                    new ObjectOutputStream( so.getOutputStream() );
                zumServer.writeBytes( objName+" ablegen \n");
                zumServer.writeObject( prd );
                zumServer.flush();

                // Abwarten der Antwort vom Server
                BufferedReader vomServer = new BufferedReader(
                    new InputStreamReader( so.getInputStream() ) );
                String antwort = vomServer.readLine();
                if( !antwort.equals("return") ) {
                    throw new Exception("Syntaxfehler in Antwort");
                }
                so.close();
            } catch( Exception e ){
                System.out.println("Fehler in Erzeuger : " + e );
                System.exit( -1 );
            }
        }
    }
}
```

Abbildung 7.16: Erzeuger als Clients

### 7.3.2 Realisierung von entfernten Methodenaufrufen in Java

Dieser Abschnitt erläutert die Implementierungstechnik, mit der Java entfernte Methodenaufrufe unterstützt. Er gibt eine allgemeine Beschreibung des RMI-Mechanismus, zeigt dessen Anwendung auf das Erzeuger-Verbraucher-Beispiel und geht detaillierter auf die Parameterübergabesemantik von RMI ein.

#### 7.3.2.1 Der Stub-Skeleton-Mechanismus

Remote-  
Objekt

Objekte, die von entfernten Prozessen aus angesprochen werden können, nennen wir *Remote-Objekte*. Bevor wir uns der Deklaration von Remote-Objekten zuwenden, betrachten wir die Implementierungstechnik, die der Realisierung von Remote-Objekten und Referenzen auf Remote-Objekte zugrunde liegt. Die Kenntnis dieser Technik ist hilfreich, um den Übersetzungsvorgang besser verstehen und den Parameterübergabemechanismus genauer beschreiben zu können. Wir erläutern die Implementierungstechnik anhand von Abb. 7.17.

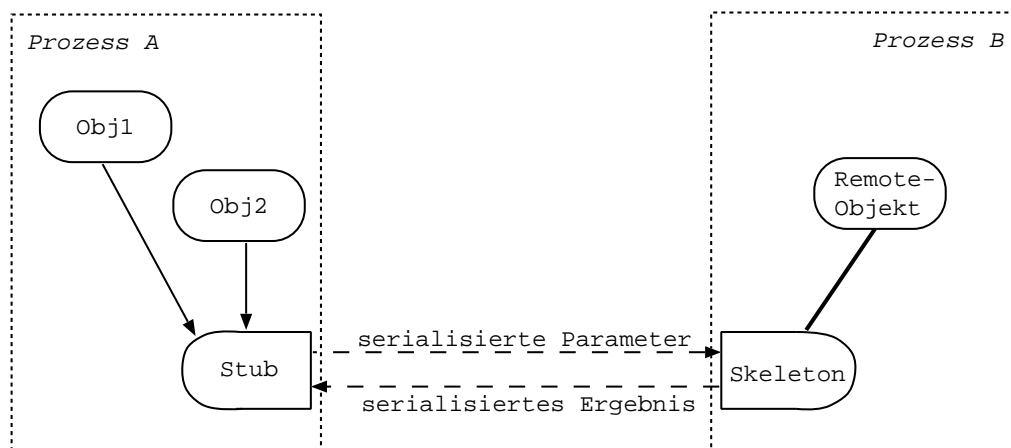


Abbildung 7.17: Realisierung von Remote-Objekten

Skeleton-  
Objekt

Jedem Remote-Objekt ist in seinem Prozess ein *Skeleton-Objekt* zugeordnet (siehe Prozess B). Das Skeleton-Objekt nimmt Methodenaufrufe und die serialisierten Parameter von anderen Prozessen entgegen. Es erzeugt für die serialisierten Parameter und die von ihnen erreichbaren Objekte Kopien im Prozess des Remote-Objekts. Dann ruft es die entsprechende Methode des Remote-Objekts mit diesen Parameterkopien auf. Nach Ausführung der Methode serialisiert es das Ergebnis und schickt es an den aufrufenden Prozess.

Stub-Objekt

Im aufrufenden Prozess (in Abb. 7.17 ist das der Prozess A) wird das Remote-Objekt von sogenannten *Stub-Objekten* repräsentiert; d.h. alle Referenzen, die logisch gesehen von Prozess A auf das Remote-Objekt verweisen, werden durch Referenzen auf ein entsprechendes Stub-Objekt realisiert. Bei

einem Methodenaufruf serialisiert das Stub-Objekt die Parameter und übernimmt die oben beschriebene Kommunikation mit dem entsprechenden Skeleton-Objekt.

Dieser Stub-Skeleton-Mechanismus ist eine übliche Realisierungstechnik für verteilte Objekte, die auch in entsprechender Form in CORBA angewandt wird. Er hat den Vorteil, dass implementierungstechnisch nicht zwischen lokalen und entfernten Referenzen unterschieden werden muss, da Referenzen auf Remote-Objekte durch lokale Referenzen auf die entsprechenden Stub-Objekte implementiert sind.

Der Programmcode für die Stub- und Skeleton-Objekte lässt sich dabei meistens automatisch von entsprechenden Werkzeugen generieren. In Java erledigt das der RMI-Compiler `rmic`: Er bekommt die Klasse des Remote-Objekts als Eingabe und erzeugt daraus die Klassen für die Stub- und Skeleton-Objekte. Ab der Version 5.0 brauchen in Java diese Klassen noch nicht einmal mehr explizit mittels `rmic` generiert werden. Zur Laufzeit werden bei Bedarf automatisch passende Klassen, sogenannte „dynamic proxies“ generiert.

### 7.3.2.2 Entfernter Methodenaufruf in Java

Jedes verteilte System besteht aus einer Menge von Prozessen (vgl. Abschn. 7.1.1). Jeder Prozess ist durch ein Programm beschrieben, das bestimmte Klassen benutzt. Grundsätzlich kann ein Prozess während seiner Laufzeit abwechselnd als Server und als Client bei entfernten Methodenaufrufen auftreten (Unterabschn. 7.3.2.3 enthält dazu ein Beispiel). Hat der Prozess mehrere Threads, kann es vorkommen, dass er sogar gleichzeitig als Server und Client agiert. Um die Aspekte von Programmen, die über entfernten Methodenaufruf kommunizieren, näher zu beschreiben, ist es aber zunächst hilfreich, so zu tun, als wären Client- und Server-Prozess jeweils durch ein Programm realisiert. Dadurch lassen sich die einzelnen Aspekte genauer und leichter ansprechen. Die Verallgemeinerung dieser vereinfachten Sicht ergibt sich dann ohne weiteres. Im Folgenden beschreiben wir zunächst die Realisierung der Server-Seite und dann den Zugriff auf Remote-Objekte von Seiten eines Clients; schließlich fassen wir die notwendigen Schritte zur Anwendung von RMI zusammen.

**Programmtechnische Behandlung von Remote-Objekten.** Da wir hier auf die Behandlung von Mechanismen zum dynamischen Laden von Programm- bzw. Klassencode verzichten, gehen wir davon aus, dass auf Client- und auf Server-Seite die Klassen bzw. Schnittstellen vorhanden sind, die sowohl im Client- wie auch im Server-Programm benutzt werden. Dies sind insbesondere die Klassen, die die Parameter-Objekte implementieren (im Erzeuger-Verbraucher-Beispiel ist das die Klasse `Produkt`). Parameter- und Ergebnis-Objekte müssen serialisierbar sein. (Remote-Objekte erben die Serialisierbar-



keit von der Klasse `RemoteObject`.)

Auf der Client-Seite muss der Zugriff auf Remote-Objekte in Java über einen Schnittstellentyp erfolgen, in dem die für den entfernten Aufruf bereitstehenden Methoden deklariert sind. Alle entfernten Methoden müssen eine `RemoteException` auslösen dürfen. Der Schnittstellentyp muss Subtyp des Schnittstellentyps `Remote` sein, der im Bibliothekspaket `java.rmi` definiert ist. Für unser Erzeuger-Verbraucher-Beispiel bedeutet das, dass es einen Schnittstellentyp `RingPuffer` mit den Methoden `ablegen` und `holen` geben muss, die beide eine `RemoteException` auslösen dürfen (vgl. die Klasse `RingPuffer` in Abb. 6.10, S. 413):

```
public interface RingPuffer extends Remote {
    public void ablegen( Produkt prd )
        throws InterruptedException, RemoteException;
    public Produkt holen()
        throws InterruptedException, RemoteException;
}
```

Wir gehen im Folgenden davon aus, dass die Klasse `Produkt` in einem Paket `produkte` deklariert ist und dass die Schnittstelle `RingPuffer` in einem Paket `pufferServer` vereinbart ist. Beide Pakete müssen auf der Client- und Server-Seite verfügbar sein.

Auf der Server-Seite muss es darüber hinaus eine Klasse geben, die die Remote-Objekte und deren Schnittstellentyp implementieren. Gemäß Javas Namenskonvention trägt diese Klasse üblicherweise den Namen des entsprechenden Schnittstellentyps mit dem Postfix „Impl“. Die ehemalige Klasse `RingPuffer` erhält also den Namen `RingPufferImpl`. Ansonsten sind nur zwei marginale Änderungen nötig, um eine Klasse mit der Fähigkeit zur Bewältigung entfernter Methodenaufrufe auszustatten:

1. Sie muss von der Klasse `UnicastRemoteObject` erben, welche im Bibliothekspaket `java.rmi.server` definiert ist (bzw. von einer anderen Klasse, die die notwendige Funktionalität bereitstellt, um ein Remote-Objekt in einem verteilten System zugreifbar zu machen).
2. Ihr Konstruktor muss eine `RemoteException` auslösen dürfen.

Der Vergleich der Klasse `RingPuffer` von Abb. 6.10 und der Klasse `RingPufferImpl` von Abb. 7.18 zeigt nochmals am Beispiel, welche Änderungen nötig sind, um den entfernten Methodenaufruf bei Objekten einer Klasse zu ermöglichen. Insbesondere sollte erkannt werden, dass die Attribut- und Methodendeklarationen davon nicht betroffen sind.

```

import java.rmi.*;
import java.rmi.server.*;
import produkte.*;

public class RingPufferServer {
    public static void main( String[] args ) throws Exception {
        RingPuffer grPuffer = new RingPufferImpl( 50 );
        RingPuffer klPuffer = new RingPufferImpl( 4 );
        Naming.rebind( "grosserPuffer", grPuffer );
        Naming.rebind( "kleinerPuffer", klPuffer );
    }
}

class RingPufferImpl extends UnicastRemoteObject
    implements RingPuffer {
    private Produkt[] puffer;
    private int eingabeIndex, ausgabeIndex;
    private int pufferGroesse;
    private int gepuffert; // Anzahl gepufferter Elemente

    public RingPufferImpl( int groesse )
        throws RemoteException {
        eingabeIndex = 0;
        ausgabeIndex = 0;
        gepuffert = 0;
        pufferGroesse = groesse;
        puffer = new Produkt[pufferGroesse];
    }

    ... // der Rest der Klasse ist identisch mit dem
        // entsprechenden Teil der Klasse RingPuffer
}

```

Abbildung 7.18: RingPuffer-Klasse für entfernten Methodenauftruf

**URLs für entfernte Objekte.** Bisher haben wir nur beschrieben, wie Remote-Objekte zu implementieren sind, und erläutert, dass die Client-Seite auf die Remote-Objekte über einen extra vereinbarten Schnittstellentyp zugreifen muss. Um den RMI-Mechanismus von Java anwenden zu können, muss noch geklärt werden, wie von der Server-Seite eine Referenz auf ein Remote-Objekt zur Verfügung gestellt werden kann und wie man auf Seiten der Clients eine solche Referenz ansprechen kann, d.h. unter welcher Bezeichnung man auf entfernte Objekte zugreifen kann (vgl. Abschn. 7.3.1).

Die Verbindung zu entfernten Objekten wird über einen Namensserver hergestellt. Der übliche Namensserver für RMI in Java heißt `rmiregistry`. Er muss auf der Server-Seite laufen, damit der RMI-Mechanismus funktionieren kann. (In den meisten Rechnerumgebungen reicht es dazu aus, das Kommando `rmiregistry` aufzurufen und ggf. dafür zu sorgen, dass der gestartete Namensserver im Hintergrund abläuft.) Der Prozess, in dem sich Remote-Objekte befinden, meldet diese mittels der statischen Methoden `bind` oder `rebind` der Klasse `Naming` aus dem Bibliothekspaket `java.rmi` unter einem beliebigen Namen beim Namensserver an. Client-Prozesse können auf angemeldete Remote-Objekte über URLs zugreifen. Der URL besteht wie üblich aus drei Teilen (vgl. S. 452): dem Protokollnamen `rmi`; der Adresse des Rechners, auf dem der Namensserver und der Server-Prozess läuft (die Portnummer ist standardgemäß 1099, vgl. Abb. 7.4); und dem Namen des Objekts.

Abbildung 7.18 zeigt das Vorgehen auf der Server-Seite: Der RingPuffer-Server erzeugt zwei Puffer, einen mit 50 und einen mit 4 Speicherplätzen. Das eine Puffer-Objekt wird beim Namensserver unter dem Namen `grosserPuffer`, das andere unter dem Namen `kleinerPuffer` angemeldet. Auf den ersten Blick könnte man meinen, dass der RingPuffer-Server nach der Anmeldung der Objekte beim Namensserver sofort wieder terminiert, so dass die Remote-Objekte ihren Service gar nicht anbieten können. Dies ist aber nicht der Fall. Remote-Objekte erben von der Klasse `UnicastRemoteObject` die Eigenschaft, dass mit Objekterzeugung ein eigener Thread für das Remote-Objekt gestartet wird. Dieser Thread hält das Remote-Objekt im Prinzip beliebig lange am Leben.

Abbildung 7.19 illustriert das Vorgehen auf der Client-Seite. Das Erzeugerprogramm erwartet als Parameter beim Start einen `rmi`-URL, z.B.:

```
rmi://elektra.fernuni-hagen.de/kleinerPuffer
```

Mittels der statischen Methode `lookup` der Klasse `Naming` besorgt es sich vom RMI-Namensserver auf dem Rechner `elektra.fernuni-hagen.de` die Referenz auf das Objekt mit Namen `kleinerPuffer`. Dabei wird insbesondere das entsprechende Stub-Objekt erzeugt (vgl. Abb. 7.17). Die Methode `lookup` liefert eine Referenz vom Typ `Remote`, die dann zum entsprechenden Schnittstellentyp zu konvertieren ist, in unserem Beispiel zum Typ

```

import java.rmi.*;
import produkte.*;
import pufferServer.*;

public class Erzeuger extends Thread {

    public static void main( String[] args ) throws Exception {
        String urlstr = args[0];
        RingPuffer p = (RingPuffer) Naming.lookup( urlstr );
        new Erzeuger(p). start();
    }

    private RingPuffer lager;

    Erzeuger( RingPuffer rp ){ lager = rp; }

    public void run(){
        while( true ){
            Produkt prd = new Produkt();
            try{
                sleep( prd.getProduktionsZeit() ); //"Produktionszeit"
                lager.ablegen( prd );
            } catch( RemoteException e ){
                System.out.println("RemoteException in Erzeuger");
                System.exit( -1 );
            } catch( InterruptedException e ){
                System.out.println("Unzulaessige Unterbrechung!");
                System.exit( -1 );
            }
        }
    }
}

```

Abbildung 7.19: Erzeuger mit entferntem Methodenaufruf

`RingPuffer`. Danach kann das Objekt verwendet werden, als würde es lokal auf der Client-Seite existieren. Dies führt insbesondere dazu, dass Klassen, die Remote-Objekte verwenden, sich im Wesentlichen nicht von Klassen unterscheiden, die lokale Objekte verwenden. Der einzige Unterschied besteht darin, dass entfernte Methodenaufrufe eine `RemoteException` erzeugen können und dass diese im Client geeignet behandelt werden muss. Beispielsweise ist der Programmtext der Erzeuger-Klasse in Abb. 7.19 identisch mit dem Programmtext der Erzeuger-Klasse von S. 410 mit dem Unterschied, dass er zusätzlich die `RemoteException` behandelt<sup>5</sup>.

**Die Schritte zur Nutzung von RMI.** Wie wir gesehen haben, sind bei der Anwendung des RMI-Mechanismus in Java eine Reihe von Aspekten zu beachten. Die folgende Aufzählung fasst diese Aspekte zusammen und gibt damit auch eine kurze Zusammenfassung dieses Unterabschnitts:

1. Der Typ, der die Schnittstelle der Remote-Objekte beschreibt, muss den Typ `Remote` aus dem Bibliothekspaket `java.rmi` erweitern. Er muss sowohl auf der Client-Seite als auch auf der Server-Seite verfügbar sein. Daraus folgt insbesondere auch, dass alle Parameter- und Ergebnistypen der entfernten Methoden auf Client- und Server-Seite zugreifbar sein müssen (im Beispiel ist das der Typ `Produkt`).
2. Die Klasse, die die Remote-Objekte realisiert, muss den Schnittstellentyp implementieren und sollte `UnicastRemoteObject` erweitern. Sie muss auf der Server-Seite verfügbar sein.
3. Aus der Klasse, die die Remote-Objekte realisiert, mußten bis zur Version 1.4 mittels des RMI-Compilers `rmic` die Klassen für die Stub-Objekte generiert werden<sup>6</sup>. Sie mußten auf der Server-Seite verfügbar sein<sup>7</sup>. Die Klassen für die Stub-Objekte mussten darüber hinaus auf der Client-Seite verfügbar sein (was auch bedeuten konnte, dass der Client sie dynamisch von anderer Seite laden konnte). Ab der Version 5.0 müssen auch die Stub-Klassen nicht mehr explizit generiert werden. Bei Bedarf (z.B. Start des Servers) werden dann statt dessen automatisch alle für den Remote-Zugriff benötigten Klassen generiert.
4. Bevor die Server- und Client-Programme gestartet werden, muss der Namensserver `rmiregistry` auf der Server-Seite laufen.

---

<sup>5</sup>Die `main`-Methode ist nur der Einfachheit halber in der Klasse `Erzeuger` vereinbart worden. Sie hätte ebenso in einer zusätzlichen Klasse deklariert werden können.

<sup>6</sup>Auf die explizite Generierung der Klasse für die Skeleton-Objekte konnte bereits ab der Version 1.2 verzichtet werden.

<sup>7</sup>Die serverseitige Existenz der Stub-Klassen wurde verlangt, damit der Server die Stub-Klassen an diejenigen Clients übertragen konnte, die über sie nicht lokal verfügen.

5. Das Server-Programm muss die Remote-Objekte erzeugen und beim Namensserver anmelden, bevor Client-Programme darauf zugreifen können. Das Server-Programm bleibt so lange verfügbar, bis kein Remote-Objekt mehr angemeldet ist.
6. Client-Programme können Remote-Objekte nutzen, solange der Namensserver und das Server-Programm laufen. Sie müssen natürlich die Rechneradresse des Servers und die Namen der entfernten Objekte kennen.

Wie aus der Zusammenstellung zu ersehen ist, reicht es aus, dass die Klassen, die die Remote-Objekte bzw. deren Skeleton-Objekte implementieren, nur serverseitig verfügbar sind. Bei den anderen Klassen eines verteilten Systems muss der Entwickler im Einzelfall entscheiden, bei welchen Prozessen der Programmcode der Klassen zugreifbar sein soll (vgl. Unterabschn. 7.3.1.1).

### 7.3.2.3 Parameterübergabe bei entferntem Methodenaufruf

In Unterabschn. 7.3.1.1 haben wir unterschiedliche Möglichkeiten vorgestellt, Objekte als Parameter bzw. Ergebnisse von entfernten Methoden zu behandeln. Hier wollen wir erläutern, wie die Parameterübergabe beim entfernten Methodenaufruf in Java realisiert ist. Die Parameterübergabesemantik des RMI unterscheidet zwischen Remote-Objekten – also Objekten der Klasse `RemoteObject` oder einer Subklasse davon – und allen anderen Objekten. Bei Remote-Objekten wird nur das serialisierte Stub-Objekt übergeben und auf der Zielseite ein entsprechendes Stub-Objekt erzeugt. Alle anderen Parameter- und Ergebnisobjekte werden zusammen mit dem an ihnen hängenden Geflecht serialisiert und auf der Zielseite in Kopie neu erzeugt. Im Folgenden werden wir diese beiden Fälle an Beispielen genauer betrachten und wichtige Eigenschaften erläutern.

**Behandlung von Remote-Objekten.** Die Semantik der Parameterübergabe von Remote-Objekten gleicht der Parameterübergabesemantik des normalen Methodenaufrufs in Java. Dies soll hier an einem Beispiel mit mehreren verteilten Systemteilen demonstriert werden. Dabei werden wir auch sehen, wie man Referenzen auf Remote-Objekte erhalten kann, die nicht bei einem Namensserver angemeldet sind.

Das Beispiel simuliert die Vermittlung von sogenannten Services durch einen Broker an anfragende Clients. Unter einem Service verstehen wir dabei Dienste, die von einem Server-Prozess angeboten werden. Wir realisieren dieses Szenario hier in sehr vereinfachter Form.

```

import java.rmi.*;
import java.rmi.server.*;
import gemeinsameTypen.ServiceLookup;
import gemeinsameTypen.Service;

public class EinServer {
    public static void main( String[] args ) {
        try {
            System.out.println("Server nimmt Arbeit auf");

            ServiceLookup remoteobj = new ServiceLookupImpl();
            Naming.rebind( "ServerObjekt", remoteobj );
            System.out.println("ServerObjekt angemeldet");
        } catch( Exception e ) {
            System.out.println("Fehler: " + e );
        }
    }
}

class ServiceLookupImpl extends UnicastRemoteObject
    implements ServiceLookup {
    public ServiceLookupImpl() throws RemoteException { }
    public Service getService( String servicetyp )
        throws RemoteException {
        if( servicetyp.equals("Count") ) {
            return new CountServiceImpl();
        } else if( servicetyp.equals("Print") ) {
            return new PrintServiceImpl();
        } else {
            throw new RemoteException("Service nicht verfuegbar");
        }
    }
}

class CountServiceImpl extends UnicastRemoteObject
    implements Service {
    public CountServiceImpl() throws RemoteException { }
    public int doService( String s ) throws RemoteException {
        return s.length();
    }
}

class PrintServiceImpl extends UnicastRemoteObject
    implements Service {
    public PrintServiceImpl() throws RemoteException { }
    public int doService( String s ) throws RemoteException {
        System.out.println( s );
        return 0;
    }
}

```

Abbildung 7.20: Ein einfacher, Dienste anbietender Server

Ein Server-Prozess, der Dienste anbieten will, meldet dazu bei seinem lokalen Namensserver ein Objekt vom Typ `ServiceLookup` an:

```
interface ServiceLookup extends Remote {
    Service getService( String s ) throws RemoteException;
}
```

Bei diesem Objekt können sich Broker oder Clients mittels der Methode `getService` eine Referenz auf ein Service-Objekt besorgen. Der String-Parameter beschreibt den Namen des gewünschten Service. Ein Service-Objekt ist ein Remote-Objekt, das im Server-Prozess läuft. Die Methode `getService` liefert also eine Referenz auf ein Remote-Objekt. Ein Service-Objekt bietet einen Dienst an, den man mit der Methode `doService` abrufen kann:

```
interface Service extends Remote {
    int doService( String s ) throws RemoteException;
}
```

Die Methode `doService` kann nur solche Dienste steuern, bei denen ein String als Parameter und ein int-Wert als Ergebnis ausreicht. Wir werden im Folgenden nur zwei fast triviale Dienste betrachten: `Count` zählt die Anzahl der Zeichen des Parameterstrings; `Print` druckt den Parameterstring auf dem Server aus. Abbildung 7.20 zeigt die Implementierung eines Servers, der die beschriebene Funktionalität in einfacher Weise bietet.

Normalerweise hat ein Broker Kenntnis von mehreren Diensteanbietern und kann deren Services an anfragende Clients nach geeigneten Optimalitätskriterien vermitteln. In unserer trivialen Variante kennt der Broker nur einen dienst anbietenden Server, an den er alle Clients vermittelt. Die Adresse des Servers wird dem Broker-Prozess beim Start als Argument mitgegeben. Abbildung 7.21 zeigt eine entsprechende Realisierung.

Bevor wir das Zusammenspiel von Client, Broker und Server und dabei insbesondere die Behandlung der Remote-Objekte erörtern können, wollen wir noch einen einfachen Client skizzieren: Er erwartet als Eingabeparameter die Adresse eines Brokers<sup>8</sup> sowie eine beliebige Zeichenreihe. Über den Broker besorgt er sich ein Service-Objekt für den `Count`-Service. Wenn die Länge der eingegebenen Zeichenreihe in eine Zeile passt, d.h. weniger als achtzig Zeichen hat, besorgt er sich darüber hinaus ein Service-Objekt für den `Print`-Service und lässt die Zeichenreihe damit zweimal drucken (vgl. Abb. 7.22). Das wiederholte Drucken soll nur andeuten, dass man einen Service selbstverständlich mehrfach in Anspruch nehmen kann.

---

<sup>8</sup>Die Funktionalität lässt es auch zu, dass sich der Client direkt beim Server anmeldet.



```

public class EinBroker {
    public static void main( String[] args ) {
        // args[0] : rmi-URL des Servers
        try {
            ServiceLookup server =
                (ServiceLookup) Naming.lookup( args[0] );
            ServiceLookup brobj = new ServiceLookupBImpl( server );
            Naming.rebind( "BrokerObjekt", brobj );
        } catch( Exception e ) {
            System.out.println("Fehler: " + e );
        }
    }
}

class ServiceLookupBImpl extends UnicastRemoteObject
    implements ServiceLookup {
    private ServiceLookup server;
    public ServiceLookupBImpl( ServiceLookup s )
        throws RemoteException {
        server = s;
    }
    public Service getService( String servicetyp )
        throws RemoteException {
        return server.getService( servicetyp );
    }
}

```

Abbildung 7.21: Ein trivialer Broker

```

public class EinClient {
    public static void main( String[] args ) {
        // args[0] : rmi-URL des Brokers
        // args[1] : Ausgabezeichenreihe
        try {
            ServiceLookup broker =
                (ServiceLookup) Naming.lookup(args[0]);
            Service count = broker.getService("Count");
            if( count.doService( args[1] ) < 80 ) {
                Service print = broker.getService("Print");
                print.doService( args[1] );
                print.doService( args[1] );
            }
        } catch( Exception e ) {
            System.out.println("Fehler: " + e );
        }
    }
}

```

Abbildung 7.22: Ein einfacher Client

Abbildung 7.23 illustriert die Verteilung der Remote- und Stub-Objekte, nachdem der Client sich den Print-Service besorgt hat. Sie verdeutlicht u.a., dass von Remote-Objekten keine Kopien angelegt werden und dass insbesondere die Service-Objekte auf dem Server verbleiben und dementsprechend auch dort ihre Dienste ausführen. Das Beispiel zeigt darüber hinaus, dass Referenzen auf Remote-Objekte in Form von Stub-Objekten von einem Systemteil zum anderen weitergereicht werden können. Im Beispiel hat der Broker

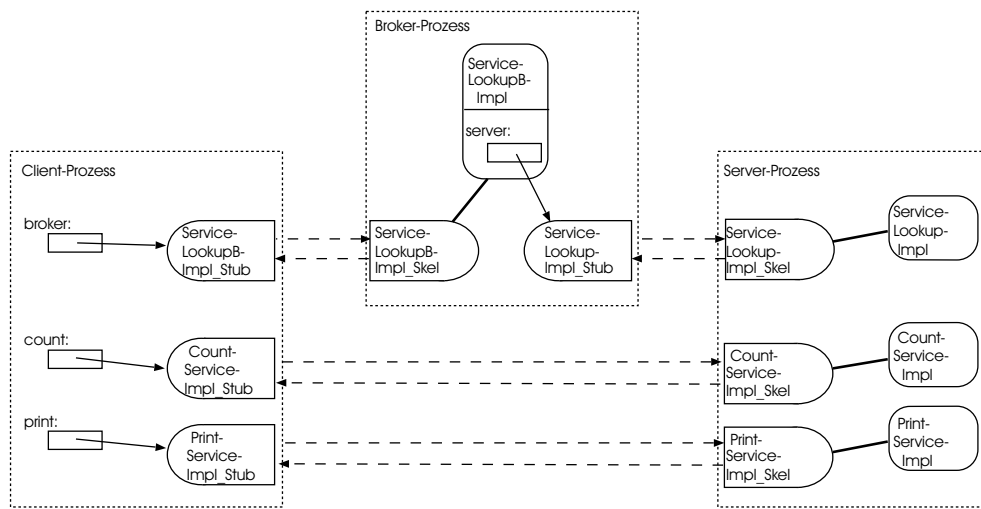


Abbildung 7.23: Einfaches Client-Broker-Server-Szenario

die „Referenzen auf die Service-Objekte übertragen“, indem er ihre serialisierten Stub-Objekte übertragen hat. Dadurch kann insbesondere ein Prozess eine Referenz auf ein Remote-Objekt in einem anderen Prozess erhalten, ohne dass dies für ihn sichtbar wird. Beispielsweise erhält der Client die Referenz eines Service-Objekts auf einem Server, von dessen Adresse und Existenz er keine Kenntnis hat. Im Wesentlichen ist es also für einen Client nicht erkennbar, wo sich das Remote-Objekt befindet, dessen Methode er aufruft. Möglicherweise liegt das Remote-Objekt sogar in seinem eigenen Prozess, d.h. er nimmt nicht einmal wahr, ob die aufgerufene Methode im eigenen Prozess oder wirklich entfernt ausgeführt wird.

**Präzisierung der Parameterübergabe.** Die wesentlichen Prinzipien der Parameterübergabe bei entferntem Methodenaufruf haben wir kennen gelernt: Entfernte Objekte werden per Referenz auf ein entsprechendes Stub-Objekt übergeben; alle anderen Objekte durch Serialisierung. Drei Fragen sind aber bisher unbeantwortet geblieben:

1. Wie wird ein Remote-Objekt behandelt, wenn es indirekt von einem Parameter referenziert wird, der per Serialisierung übergeben wird?

2. Wird ein Remote-Objekt eines entfernten Prozesses im lokalen Prozess immer durch genau ein Stub-Objekt repräsentiert oder gibt es in einem Prozess ggf. mehrere Stub-Objekte, die dasselbe Remote-Objekt repräsentieren?
3. Wenn ein Objekt bei einem entfernten Methodenaufruf von mehr als einem aktuellen Parameter direkt oder indirekt referenziert wird, wird von diesem Objekt dann eine oder werden mehrere Kopien im entfernten Prozess erzeugt?

Die Antwort auf die erste Frage entspricht der Erwartung: Remote-Objekte werden immer per Referenz auf ein Stub-Objekt übergeben, insbesondere auch dann, wenn sie innerhalb eines Objektgeflechts referenziert werden, das zu einem zu serialisierenden Parameter gehört.

Die Antwort auf die zweite Frage ist, dass es in einem Prozess viele Stub-Objekte geben kann, die dasselbe Remote-Objekt repräsentieren. Es kann auch vorkommen, dass in einem Prozess sowohl ein Remote-Objekt als auch ein korrespondierendes Stub-Objekt existieren; nämlich dann, wenn die Referenz auf das Remote-Objekt an den Prozess von einem entfernten Methodenaufruf zurückgeliefert wurde. Liefert also ein Vergleich zweier Stub-Objekte mit dem Gleichheitsoperator „==“ den Wert `false`, kann man nicht schließen, dass sie verschiedene Remote-Objekte repräsentieren. Zu diesem Zweck kann man sich der Methode `equals` bedienen: Sie ist in der Klasse `RemoteObject` so definiert, dass sie auf Identität des zugrunde liegenden Remote-Objekts prüft.

Zur Beantwortung der dritten Frage betrachten wir zunächst folgendes Beispiel mit dem entfernten Methodenaufruf in der durch (+) markierten Zeile. Das Remote-Objekt spielt hier keine Rolle, wir konzentrieren uns ganz auf die drei aktuellen Parameter des Methodenaufrufs. Sie sind alle vom Typ `EinTyp`, können also über ihr Attribut `aenderemich` ein `StringBuffer`-Objekt referenzieren:

```
interface RemoteTyp extends Remote {
    StringBuffer machwas( EinTyp t1, EinTyp t2, EinTyp t3 )
        throws RemoteException;
}

public class EinTyp implements Serializable {
    public StringBuffer aenderemich;
    public EinTyp( StringBuffer sb ){ aenderemich = sb; }
}

public class EinTest {
    public static void main( String[] args ) {
        RemoteTyp remoteobj = ...
    }
}
```

```

...
StringBuffer sb = new StringBuffer("MODE");
EinTyp et1      = new EinTyp( sb );
EinTyp et2      = new EinTyp( sb );
(+) sb = remoteobj.machwas(et1,et2,et2);
...
}
}

```

Abbildung 7.24 zeigt das von den drei aktuellen Parametern referenzierte Objektgeflecht: Der erste aktuelle Parameter ist eine Referenz auf ein Objekt *o1*; der zweite und dritte aktuelle Parameter sind gleich und referenzieren das Objekt *o2*. Die Objekte *o1* und *o2* referenzieren das StringBuffer-Objekt *o3*. Wie

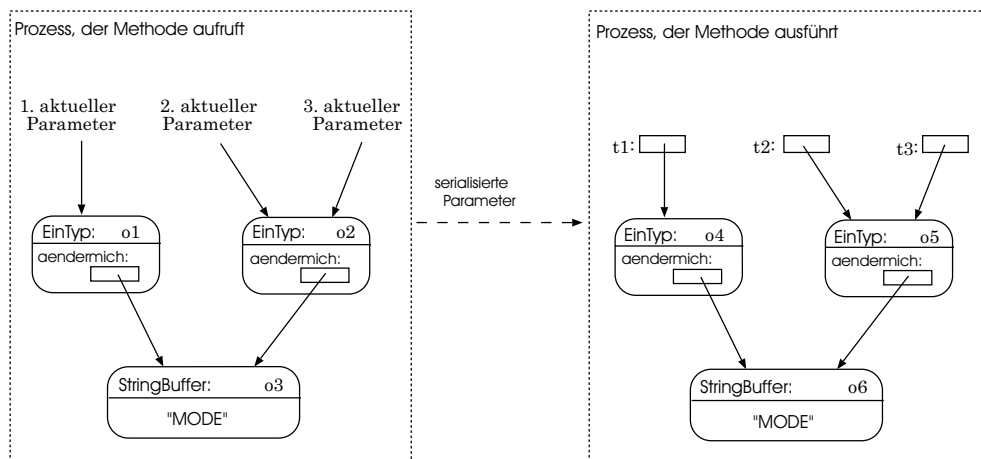


Abbildung 7.24: Ein einfaches Objektgeflecht mit drei Einstiegsreferenzen

in Abb. 7.24 illustriert, werden bei der Parameterübergabe in dem entfernten Prozess, der die Methode ausführt, Kopien *o4*, *o5* und *o6* dieser drei Objekte erzeugt und entsprechend verzeigert; d.h. insbesondere, dass die Parameter nicht einzeln serialisiert werden und dass deshalb gleiche Parameter nicht doppelt serialisiert werden (zur Serialisierung mehrerer Objekte vgl. die Diskussion am Ende von Unterabschn. 4.3.2.2, S. 281). Dadurch wird zumindest die „relative“ Objektidentität zwischen denjenigen Objekten gewahrt, die innerhalb des Objektgeflechts liegen, das von den Parametern referenziert wird. Dies gilt allerdings nur im Rahmen *eines* entfernten Methodenaufrufs. Wird die gleiche Methode mit den gleichen Parametern ein zweites Mal aufgerufen, wird auf Seiten des entfernten Prozesses eine weitere Kopie des Objektgeflechts erzeugt. Entsprechendes gilt für Objektgeflechte, die von Ergebnisobjekten referenziert werden.



# Kapitel 8

## Zusammenfassung, Varianten, Ausblick

Dieses Kapitel bietet eine Zusammenfassung der objektorientierten Konzepte und geht in Kürze auf wichtige Aspekte objektorientierter Sprachen ein, die in Java keine Berücksichtigung fanden. Schließlich skizziert es mögliche zukünftige Entwicklungslinien. Insgesamt möchte dieses abschließende Kapitel nochmals hervorheben, dass objektorientierte Programmierung nicht an einer Programmiersprache hängt, sondern aus der Anwendung einer Reihe von Konzepten besteht, die in verschiedenen Sprachen oft aus guten Gründen unterschiedlich ausgeprägt sind und unterstützt werden. Darüber hinaus soll an einigen Aspekten deutlich gemacht werden, dass objektorientierte Programmierung sowohl in der softwaretechnischen Realisierung als auch in der Informatikforschung noch lange kein abgeschlossenes Kapitel ist.

### 8.1 Objektorientierte Konzepte zusammengefasst

Objektorientierte Programmierung hat zwei Seiten (vgl. Einleitung): Einerseits die Anwendung objektorientierter Konzepte in der Programmierung, andererseits die Umsetzung objektorientierter Konzepte in Konstrukte von Programmiersprachen. Dieser Abschnitt bietet eine Zusammenfassung der wesentlichen Konzepte objektorientierter Programmierung. Realisierungsvarianten dieser Konzepte in unterschiedlichen Programmiersprachen sind Gegenstand von Abschn. 8.2.

Die wissenschaftliche Diskussion darüber, was die wesentlichen objektorientierten Konzepte sind und welche Merkmale eine Programmiersprache besitzen muss, um als objektorientiert zu gelten, ist nicht abgeschlossen (als Einstieg in die Diskussion ist [Weg87] geeignet). Insbesondere werden die Konzepte unterschiedlich gewichtet und strukturiert.

Wir vertreten hier die folgende Sichtweise. Die objektorientierte Programmierung basiert auf zwei *Grundkonzepten*, nämlich dem Objektkon-

zept und der Klassifikation. Bei der programmiersprachlichen Umsetzung dieser Grundkonzepte stehen drei sprachliche Konzepte im Mittelpunkt, nämlich Klassen, Vererbung und dynamische Methodenauswahl; bei typisierten Sprachen spielt auch die Realisierung von Subtyping eine wichtige Rolle. Zunächst sollen die beiden Grundkonzepte zusammengefasst werden:

- **Objektkonzept:** Softwaresysteme werden als eine Menge kooperierender Objekte modelliert. Konzeptionell sind Objekte eigenständige Ausführungseinheiten, die selbst Berechnungen durchführen und auf Nachrichten anderer Objekte reagieren können. Programmtechnisch heißt das, dass Objekte ihre eigenen Operationen und ihren eigenen Zustand besitzen (wirklich aktive Objekte werden darüber hinaus einen eigenen Ausführungsstrang haben, ähnlich den Java-Threads; vgl. die Diskussion in Abschn. 6.1.2). Die Ausführung eines objektorientierten Programms besteht dann im Nachrichtenaustausch zwischen Objekten, den Zustandsänderungen, die Objekte an sich selbst vornehmen, und dem Erzeugen neuer Objekte (bzw. dem Löschen von Objekten). Im Vergleich zur prozeduralen Programmierung führt das Objektkonzept
  1. zu einer anderen Programmstruktur: Objekte fassen Daten, Zustand und zugehörige Operationen in (kleinen) Einheiten zusammen, während die prozedurale Programmierung Daten, Zustand und Operationen konzeptionell trennt;
  2. zu einem anderen Ausführungsmodell: das in Kapitel 1.2.3 bereits eingeführte objektorientierte Grundmodell unterstellt ein inhärent paralleles Ausführungsmodell und lässt sich deshalb gut für die verteilte Programmierung verwenden<sup>1</sup>; die prozedurale Programmierung ist sequentiell angelegt.

Wichtige Konsequenz des Objektkonzepts ist eine saubere Schnittstellenbildung. Die Benutzungsschnittstelle eines Objekts besteht aus den öffentlich verfügbaren Nachrichten und den öffentlich zugreifbaren Attributen. Die Schnittstelle abstrahiert von der Implementierung der Objekte. Sie verbirgt die Implementierung vor den Benutzern. Dadurch werden Implementierungen austauschbar, die ein äquivalentes Verhalten an der öffentlichen Schnittstelle aufweisen.

- **Klassifikation:** Objekte lassen sich gemäß ihrer öffentlichen Schnittstelle klassifizieren. Die Klassifikation bietet die Grundlage für Abstraktion und Spezialisierung, zwei zentrale Techniken der Programmentwicklung:

---

<sup>1</sup>Wie wir aber auch bereits in Kapitel 1.2.3 angemerkt haben, bieten viele der objektorientierten Programmiersprachen diese inhärente Parallelität **nicht**. Wie in der prozeduralen Programmierung bedarf es in diesen Sprachen expliziter Programmkonstrukte (wie z.B. der Threads in Java), um parallele Ausführungsstränge zu erzeugen.

1. Mittels Abstraktion kann man gemeinsame Eigenschaften unterschiedlicher Objekte oder Typen zu einem abstrakteren Typ zusammenfassen (Verkleinern der Schnittstelle); Programme, die sich nur auf diese gemeinsamen Eigenschaften stützen, arbeiten dann für alle Objekte des abstrakteren Typs.
2. Mittels Spezialisierung kann man Objekte bzw. Typen und ihre Implementierung erweitern, d.h. ihnen zusätzliche und speziellere Eigenschaften geben. Wir gehen dabei davon aus, dass sich spezialisierte Objekte konform zu den allgemeineren Objekten verhalten. Programme, die auf den allgemeineren Objekten gearbeitet haben, arbeiten dann auch korrekt auf den spezielleren Objekten. Spezialisierung geht sehr häufig Hand in Hand mit dem Vererben von Implementierungsteilen und besitzt damit große Bedeutung für die Software-Wiederverwendung.

Die *sprachliche Realisierung* dieser beiden Grundkonzepte basiert im Wesentlichen auf drei Sprachkonzepten, die sich in der einen oder anderen Form in den meisten objektorientierten Programmiersprachen wiederfinden:

- **Klassenkonzept:** Klassen dienen zur Deklaration der Eigenschaften von Objekten. Klassen verbinden Typisierungs- und Modularisierungsaspekte. Insbesondere unterstützen sie eine saubere Schnittstellenbildung für ihre Objekte und eine flexible Kapselung von Implementierungsteilen.
- **Vererbungskonzept:** Vererbung ermöglicht es, Eigenschaften existierender Objekte bzw. Klassen für die Deklaration neuer Objekte bzw. Klassen zu verwenden. Dadurch wird
  - fehlerträchtiges Kopieren von Programmtexten vermieden,
  - der Umfang der Programme durch mehrfache Verwendung innerhalb der Vererbungshierarchie verringert und
  - eine flexible Anpassung bzw. Erweiterung von Systemschnittstellen, bei denen Kopieren und Modifizieren nicht realisierbar ist, ermöglicht.
- **Dynamische Methodenauswahl und Subtyping:** In einem prozeduralen Programm ist bei einem Aufruf  $p(x_0, x_1, \dots, x_n)$  zur Übersetzungszeit bekannt, welche Prozedur ausgeführt wird. In einem objektorientierten Programm wird erst dynamisch, also zur Laufzeit, entschieden, welche Methode bei einer Anweisung  $x_0.m(x_1, \dots, x_n)$  ausgeführt wird: Das Objekt *obj*, das von der Variablen  $x_0$  referenziert wird, „entscheidet“ darüber, welche Methode zur Ausführung gelangt (vgl. Kap. 3). Deshalb sagt man auch, dass *obj* die Nachricht  $m(x_1, \dots, x_n)$



geschickt wird und *obj* die auszuführende Methode bestimmt (vgl. Objektkonzept oben). Da die Auswertung des Ausdrucks  $x_0$  zu unterschiedlichen Ausführungszeitpunkten im Allgemeinen Objekte mit unterschiedlichen Eigenschaften als Ergebnis hat, kommen durch die Anweisung  $x_0.m(x_1, \dots, x_n)$  immer wieder andere Methoden zur Ausführung.

Bei einer typisierten Sprache kann dynamische Methodenauswahl nur realisiert werden, wenn Variablen (wie  $x_0$ ) Objekte unterschiedlichen Typs referenzieren dürfen. Um die Konzepte der objektorientierten Programmierung in typisierten Sprachen anwenden zu können, benötigt man deshalb Subtyping.

Der folgende Abschnitt erläutert anhand einiger Beispiele, wie diese Sprachkonzepte innerhalb unterschiedlicher objektorientierter Sprachen umgesetzt wurden. Dabei wird u.a. gezeigt, dass diese Sprachkonzepte weitgehend unabhängig sind; das heißt, dass sich jedes Konzept unabhängig von den anderen realisieren lässt. Die Attraktivität der Objektorientierung ergibt sich allerdings gerade aus dem harmonischen Zusammenspiel der Konzepte.

## 8.2 Varianten objektorientierter Sprachen

Objektorientierte Programmiersprachen unterscheiden sich darin, wie sie die obigen Konzepte in Sprachkonstrukte umsetzen und mit anderen Sprachkonzepten und -konstrukten verbinden. Wir betrachten hier zunächst objektorientierte Erweiterungen prozeduraler Sprachen und dann Sprachen, die speziell für die objektorientierte Programmierung entwickelt wurden. Darüber hinaus sind eine Reihe von Sprachen entwickelt worden, die Objektorientierung mit anderen als prozeduralen Sprachkonzepten verbinden; z.B. ist CLOS eine objektorientierte Erweiterung der funktionalen Sprache Common Lisp ([Kee89]) und Oz ([Smo95]) eine Sprache zur Constraint-basierten objektorientierten Logikprogrammierung.

### 8.2.1 Objektorientierte Erweiterung prozeduraler Sprachen

Typische Vertreter dieser Sprachklasse sind C++, Ada95 und Modula-3. Hauptvorteil derartiger Erweiterungen ist, dass Programme, die in der Originalsprache (hier also C, Ada, Modula-2) geschrieben wurden, ohne bzw. mit wenigen Änderungen weiter benutzt werden können. Außerdem bieten diese Sprachen mit ihren prozeduralen Sprachkonstrukten mehr Flexibilität und die Möglichkeiten einer maschinennäheren Programmierung an, insbesondere auch die weitgehende Kontrolle über Speicherressourcen (dazu gehört zum Beispiel die Möglichkeit, den Speicher von Objekten freigegeben

zu können, wenn das Objekt nicht mehr gebraucht wird). Dies ist für zeitkritische, systemnahe Anwendungen nach wie vor unerlässlich. Andererseits entstehen durch die Hinzunahme objektorientierter Sprachkonstrukte sehr umfangreiche, in ihrer Gesamtheit oft schwer zu beherrschende Programmiersprachen, die darüber hinaus die objektorientierten Konzepte nur unvollständig unterstützen. Als Beispiel für eine objektorientierte Erweiterung einer prozeduralen Sprache betrachten wir hier die Programmiersprache C++ ([Lip91] bietet eine gute Einführung in die Programmierung mit C++; [ES90] enthält den annotierten Sprachbericht und [Str00] ist quasi das Standardwerk zu C++).

**Objekte und ihre Verwaltung in C++.** C++ erweitert die Programmiersprache C<sup>2</sup>. Wie bei den meisten objektorientierten Erweiterungen ist das Objektkonzept in C++ relativ schwach ausgeprägt; insbesondere kann man keine Objekte realisieren, die eigene Ausführungsstränge definieren können. Ein Objekt ist ein Verbund (engl. record), dessen Komponenten Daten und Methoden sind (C++-Sprechweise: data members, member functions). C++ macht einen expliziten Unterschied zwischen Objekten und Zeigern auf Objekte<sup>3</sup>. Rekursive Abhängigkeiten zwischen Typen müssen über Zeigertypen aufgelöst werden. Um diesen Unterschied zwischen Java und C++ zu demonstrieren, andererseits aber auch die syntaktische Ähnlichkeit von Java und C++ zu illustrieren, betrachten wir eine Klassendefinition für einfach verkettete int-Listen in Java (links) und C++ (rechts). Die letzte Zeile zeigt die Anwendung der entsprechenden Konstruktoren:

<pre>class List {     int    head;     List  tail; public     List( int e, List l ){         head = e;         tail = l;     } } ... List mylist =     new List(71,null);</pre>	<pre>class List {     int    head;     List* tail; public:     List( int e, List* l ){         head = e;         tail = l;     } }; ... List* mylist =     new List(71,NULL);</pre>
---	---

Zeigertypen werden in C++ mit einem „\*“ notiert, d.h. List\* ist der Typ der Zeiger auf List-Objekte bzw. auf Objekte der Subklassen von List; er entspricht damit dem Typ List in Java (vgl. Abschn. 1.3.2).

<sup>2</sup>Eine andere objektorientierte Erweiterung von C ist Objective-C.

<sup>3</sup>Darüber hinaus gibt es in C++ noch sogenannte „references“. Sie stellen eine Verallgemeinerung von Variablenparametern dar und haben dementsprechend einige Aspekte mit Zeigern gemeinsam, unterstützen aber ein anderes, eingeschränktes Zugriffsverhalten.

Speicher-  
bereinigung

Im Unterschied zu Java unterstützt C++ keine automatische *Speicherbereinigung* (engl. *garbage collection*). Während in Java die Laufzeitumgebung dafür sorgt, dass der Speicher nicht mehr erreichbarer Objekte wieder freigegeben und für neu zu erzeugende Objekte zur Verfügung gestellt wird, muss sich in C++ der Programmierer um die Freigabe des Speichers kümmern. Dafür kann er sogenannte Destruktoren definieren und verwenden. Die Speicher-verwaltung durch den Programmierer, wie sie auch in den meisten prozeduralen Sprachen üblich ist, hat den Vorteil, dass der Speicher effizienter genutzt werden kann. Andererseits ist die Verwaltung des Speichers, insbesondere im Zusammenhang mit objektorientierter Programmierung, eine recht komplexe Aufgabe, die den Programmieraufwand zum Teil deutlich steigert und nicht selten zu schwerwiegenden Programmfehlern führt.

**Klassenkonzept.** Das Klassenkonzept von C++ ähnelt in vielen Aspekten demjenigen von Java. Insbesondere unterstützt C++ ebenfalls innere Klassen. Allerdings ist die Semantik der inneren Klassen und der Zugriffsmodifikatoren in den beiden Sprachen zum Teil recht verschieden. Im Übrigen bietet C++ weitere Konstrukte an, um die Zugriffsrechte zwischen Klassen zu regeln. Der größte Unterschied beider Realisierungen besteht darin, dass C++ Mehrfachvererbung unterstützt, dafür aber keine Schnittstellentypen (zum Zusammenhang zwischen Mehrfachvererbung und mehrfachem Subtyping mit Schnittstellentypen vgl. Abschn. 3.3.2).

Mehrfach-  
vererbung

*Mehrfachvererbung* bedeutet, dass eine Klasse die Attribute und Methoden von mehreren Superklassen (in C++-Sprechweise: Basisklassen) erben kann. Dies erlaubt zwar für bestimmte Probleme recht elegante Lösungen, führt aber oft auch zu komplexeren und schwerer verständlichen Programmen. Insbesondere können Namenskonflikte auftreten und komplexe Vererbungsbeziehungen entstehen:

- Was soll es bedeuten, wenn zwei Superklassen gleich benannte Attribute haben; besitzt die abgeleitete Klasse das Attribut dann einmal oder zweimal und wie kann darauf zugegriffen werden?
- Was soll es bedeuten, wenn Klasse D von zwei Klassen B und C erbt, wobei sowohl B als auch C von der Klasse A abgeleitet sind? Sollen die Attribute von A dann doppelt in D vorkommen oder nur einmal? Da je nach Umfeld beide Möglichkeiten sinnvoll sein können, kann der Programmierer in C++ wählen. Dadurch werden größere Klassenhierarchien aber schnell undurchschaubar.

**Subtyping und Methodenauswahl.** Sieht man von Klassenmethoden ab, werden in Java Methoden grundsätzlich dynamisch gebunden. C++ lässt dem Programmierer die Wahl, ob Methoden statisch (Normalfall) oder dynamisch (Schlüsselwort: `virtual`) gebunden werden sollen. Allerdings unterstützt C++ Subtyping und damit dynamische Methodenauswahl nur bei Zeigertypen; d.h. Zeiger auf Objekte einer Subklasse B können ohne Typkonvertierung an Zeigervariablen einer Superklasse A zugewiesen werden, also an Variablen vom Typ A\* (für reference-Types in C++ gilt Entsprechendes). Bei der Zuweisung von B-Objekten an Variablen vom Typ A wird eine implizite Konvertierung vorgenommen, die das B-Objekt in ein A-Objekt umwandelt. Dabei wird das Objekt kopiert, und es werden insbesondere alle Attribute entfernt, die nur in den Subklassenobjekten existieren. Dies demonstrieren wir anhand der folgenden Klassen A und B, wobei B eine Subklasse von A ist und die Methode `print` überschreibt:

```
class A {
public:
    virtual void print() {
        printf("Ich bin ein A-Objekt\n");
    }
};

class B : public A {          // B erbt von A
public:
    virtual void print() {
        printf("Ich bin ein B-Objekt\n");
    }
};

(1) B* myBptr = new B();
(2) A* myAptr = myBptr;
(3) A myAobj = *myBptr;      // *myBptr bezeichnet das Objekt,
                             // auf das myBptr zeigt.
(4) A* thyAptr = &myAobj;    // laesst thyAptr auf das Objekt
                             // in Variable myAobj zeigen

(5) (*myAptr).print();       // druckt: "Ich bin ein B-Objekt"
(6) myAobj.print();          // druckt: "Ich bin ein A-Objekt"
(7) (*thyAptr).print();      // druckt: "Ich bin ein A-Objekt"
```

Die obigen Anweisungen illustrieren die komplexere Semantik von C++ im Vergleich zu Java. In Zeile (1) wird ein B-Objekt erzeugt und der Zeiger auf dieses Objekt der Zeigervariablen `myBptr` zugewiesen. In Zeile (2) wird der Zeiger auf das B-Objekt auch an die Zeigervariable `myAptr` zugewiesen. Da C++ Subtyping auf Zeigertypen zulässt, ist diese Zuweisung mit keiner Umwandlung verbunden. In Zeile (3) wird das von `myBptr` referenzierte Objekt

an die Objektvariable `myAobj` zugewiesen. Weil C++ auf Objekttypen kein Subtyping unterstützt, wird das Objekt dabei in ein A-Objekt umgewandelt. In Zeile (4) wird die Zeigervariable `thyAptr` deklariert und ihr der Zeiger auf das A-Objekt in `myAobj` zugewiesen. Zeile (5) demonstriert einen dynamisch gebundenen Methodenaufruf: Da `myAptr` ein B-Objekt referenziert, kommt es zur angegebenen Ausgabe. Die Aufrufe in den Zeilen (6) und (7) zeigen, dass das B-Objekt bei der Zuweisung an `myAobj` wirklich in ein A-Objekt umgewandelt wurde, sodass sich das Objekt in beiden Fällen als A-Objekt zu erkennen gibt.

Zusammenfassend kann man feststellen, dass C++ die wesentlichen objektorientierten Konzepte sprachlich unterstützt. Die Integration mit den prozeduralen Sprachelementen und Zeigerdatentypen führte allerdings zu teilweise recht komplexen Lösungen. Dies schafft ein erhöhtes Fehlerpotential und erschwert tendenziell die Lesbarkeit der Programme. Dazu trägt auch die in C++-Programmen häufig anzutreffende Mischung aus prozeduralen und objektorientierten Programmiertechniken und -stilen bei. Andererseits ist C++ für Programmierexperten eine sehr mächtige Sprache, die es gestattet, gleichzeitig mit verschiedenen Abstraktionsebenen zu arbeiten, ohne an den entscheidenden Programmstellen auf hohe Effizienz verzichten zu müssen. Darüber hinaus bieten die meisten Systeme – dem großen Verbreitungsgrad von C++ entsprechend – eine gute Programmierschnittstelle zu C++ an.

## 8.2.2 Originär objektorientierte Sprachen

Die Entwicklung der objektorientierten Programmierung und ihrer Sprachen begann mit den verschiedenen Versionen der Sprache Simula, die in den sechziger Jahren am norwegischen Computing Center in Oslo entworfen und implementiert wurde. Seitdem wurden etliche Sprachen mit dem Ziel entwickelt, die objektorientierte Programmierung so gut wie möglich zu unterstützen. Wir teilen diese originär objektorientierten Sprachen in die Gruppe der typisierten und die der untypisierten Sprachen ein und erläutern einige Eigenschaften von wichtigen Vertretern dieser Gruppen.

### 8.2.2.1 Typisierte objektorientierte Sprachen

Diese Sprachgruppe umfasst u.a. Simula, deren Nachfolger BETA, Eiffel und deren Nachfolger, wie z.B. Sather, und Java. Um weitere Varianten bei objektorientierten Sprachen kennenzulernen, betrachten wir interessante Sprachaspekte von Sather und BETA.

**Trennung von Vererbung und Subtyping.** In den meisten typisierten objektorientierten Programmiersprachen fällt die Vererbungshierarchie mit der

Subtyphierarchie zusammen oder ist zumindest ein Teil von ihr (wie in Java, vgl. Kap. 3). D.h. wenn die Klasse B von der Klasse A erbt, ist B auch ein Subtyp von A. In Sather sind die beiden Hierarchien unabhängig voneinander. Auf Java übertragen, bedeutet diese Entwurfsentscheidung, dass nur an den Blättern der Subtyphierarchie Klassen stehen und alle anderen Typen als Schnittstellentypen vereinbart sind; d.h. innerhalb der Subtyphierarchie werden weder Attribute noch Methodenimplementierungen vererbt. Vererbung geschieht dann mittels gesonderter Sprachkonstrukte, die angeben, von welcher Klasse welche Attribute und welche Methoden geerbt werden sollen und wie diese ggf. umzubenennen sind. Dadurch wird es insbesondere auch möglich, dass eine Klasse B von einer Klasse A erbt, ohne dass B eine Spezialisierung von A ist (d.h. ein B-Objekt *ist kein* A-Objekt).

**Vereinheitlichung von Klassen und Methoden.** In der Sprache BETA ist ein Objekt in der Regel aktiv; d.h. es besitzt außer seinen Attributen einen Aktionsteil, der bei seiner Erzeugung ausgeführt wird. Wenn der Aktionsteil sequentiell ausgeführt wird, kann man sich den Aktionsteil wie einen Konstruktor vorstellen; der Aktionsteil kann aber auch als Koroutine oder parallel zum aktuellen Ausführungsstrang ausgeführt werden, quasi als Thread. Der folgende Programtext zeigt den sequentiellen Fall. Er deklariert ein integer-Paar-Objekt `pobj`, das seine Attribute `a` und `b` mit eins bzw. zwei initialisiert:

```
pobj : @(#  a: @integer;  b: @integer
          do 1 -> a;
            2 -> b
          #)
```

Eine derartige Objektdeklaration entspricht in Java der Deklaration einer unveränderlichen Variablen, die mit dem Objekt einer anonymen Klasse initialisiert wird (vgl. Abschn. 3.2.6.4, S. 205).

Das Klassenkonzept realisiert BETA über sogenannte *Patterns*. Patterndeclarationen sehen wie Objektdeklarationen aus, nur dass der Klammeraffe nach dem Doppelpunkt fehlt. Beispielsweise könnte man statt der obigen Objektdeklaration erst ein Pattern `Paar` deklarieren und dieses dann für die Deklaration mehrerer Paar-Objekte nutzen. In dieser eingeschränkten Form entspricht das Pattern einer Klassendeklaration mit den Attributen `a` und `b` sowie einem Konstruktor zur Initialisierung:

*Pattern*

```
Paar :  (#  a: @integer;  b: @integer
          do 1 -> a;
            2 -> b
          #)
pobj  : @Paar;
pobj1 : @Paar;
...
```

Pattern können in BETA auch als Komponenten anderer Pattern deklariert werden. Sie können dabei zur Realisierung unterschiedlicher Programmelemente verwendet werden, die man in anderen objektorientierten Sprachen durch verschiedene Sprachkonstrukte beschreiben würde. Als Beispiel fügen wir zu Paar die Patternkomponente vertauschen hinzu:

```

Paar :  (#  a: @integer;  b: @integer;
          vertauschen: (#  tmpvar: @integer
                        do a->tmpvar; b->a; tmpvar->b
                        #)
          do 1 -> a; 2 -> b
          #)

```

Das Pattern `vertauschen` beschreibt hier eine Methode mit einer lokalen Variablen `tmpvar` und dem Rumpf `a->tmpvar; b->a; tmpvar->b`. Ein Aufruf `&pobj.vertauschen` erzeugt zunächst ein `vertauschen`-Objekt, das einer Methodeninkarnationen entspricht, und führt dann den Rumpf für das Objekt `pobj` aus, d.h. vertauscht die Werte der Attribute `a` und `b` von `pobj`. Selbstverständlich unterstützt BETA auch die Parameterübergabe an solche Methoden.

Das kleine Beispiel sollte eine Idee davon geben, wie es BETA durch die Realisierung von aktiven Objekten und dem mächtigen Patternkonstrukt gelingt, *Klassen und Methoden zu vereinheitlichen*. Ausgehend von dieser Vereinheitlichung unterstützt BETA weitere sehr mächtige Konstrukte, wie z.B. Variablen für Pattern und Methoden höherer Ordnung<sup>4</sup>.

### 8.2.2.2 Untypisierte objektorientierte Sprachen

Simula war zwar die erste objektorientierte Sprache. Der Durchbruch objektorientierter Techniken kam allerdings mit Smalltalk, einer untypisierten, interpretierten Sprache, die insbesondere auch wegen ihrer für die damalige Zeit fast revolutionären, graphisch interaktiven Programmierungsumgebung großes Aufsehen erzielte. Smalltalk ist eine sehr kleine Sprache mit wenigen syntaktischen Konstrukten, in der die objektorientierten Konzepte in radikaler Weise umgesetzt sind: In Smalltalk ist fast alles ein Objekt, und fast alle Aktionen werden über Nachrichten ausgelöst. So werden zum Beispiel auch Klassen als Objekte modelliert, die auf die Nachricht `new` ein neues Objekt dieser Klasse liefern. Durch Schicken der Nachricht `addSubclass` mit einem Parameter, der eine Klasse `K` repräsentiert, kann man ein neu erzeugtes Klassenobjekt der Klasse `K` als Subklasse unterordnen. Selbst Anweisungsblöcke sind in Smalltalk Objekte, die man durch Schicken der Nachricht `value` auswerten kann.

<sup>4</sup>Eine Methode höherer Ordnung ist eine Methode, die andere Methoden als Parameter und Ergebnis haben kann.

Kontrollstrukturen werden in Smalltalk durch Nachrichten realisiert. Zum Beispiel versteht die Klasse `Boolean` die Nachricht `ifTrue:ifFalse:..` Da mehrstellige Nachrichten in Infix-Schreibweise notiert werden, erhält eine bedingte Anweisung mit Bedingung `B` in Smalltalk die Form

```
B ifTrue: Block1 ifFalse: Block2
```

wobei `B`, `Block1` und `Block2` Block-Objekte sind. Das `true`-Objekt reagiert auf die Nachricht `ifTrue: Block1 ifFalse: Block2` mit Auswertung von `Block1`, das `false`-Objekt mit Auswertung von `Block2`.

Smalltalk ist untypisiert, d.h. Variablen haben keinen Typ und man kann beliebige Objekte an sie zuweisen. Die Kontrolle darüber, dass die Objekte die Nachrichten verstehen, die man ihnen schickt, liegt ausschließlich beim Programmierer. Er wird darin nicht durch ein Typsystem bzw. vom Übersetzer unterstützt, wie in typisierten Sprachen. Ist `myvar` eine Variable, dann ist z.B. die folgende Anweisungssequenz syntaktisch korrekt, die `myvar` erst 7, dann die Zeichenreihe "untypisierte Programmierung: Juchee" zuweist und ihr schließlich die angegebene Nachricht schickt:

```
myvar := 7 .
myvar := 'untypisierte Programmierung: Juchee' .
myvar ifTrue: [ myvar := 0 ] ifFalse: []
```

Allerdings wird die Auswertung der letzten Zeile zu einem Laufzeitfehler führen, weil Zeichenreihen-Objekte die Nachricht `ifTrue:ifFalse:` nicht verstehen.

Ausgehend von Smalltalks Realisierung untypisierter objektorientierter Programmierung wurden neuere Programmiersprachen entwickelt, die effizientere Implementierungen ermöglichen bzw. noch mehr Flexibilität gestatten. So stellt beispielsweise die Sprache Cecil sogenannte Multi-Methoden zur Verfügung. Bei Multi-Methoden erfolgt die dynamische Methodenauswahl nicht nur abhängig vom impliziten Parameter, sondern auch in Abhängigkeit weiterer Parameter. Ein anderes Beispiel für mehr Flexibilität liefert die Sprache Self. Self kennt keine Klassen; Objekte werden durch Klonen<sup>5</sup> existierender Objekte erzeugt, erben deren Eigenschaften und können dynamisch durch Hinzufügen neuer Attribute und Methoden verändert werden. In Self ist es auch möglich, die Vererbungsbeziehung zwischen Objekten während der Laufzeit zu verändern (ein Objekt kann sozusagen seine Eltern wechseln).

<sup>5</sup>Sprachen, die Objekte auf diese Weise erzeugen, werden *prototypbasiert* genannt; vgl. Abschn. 2.1.1.



### 8.3 Zukünftige Entwicklungslinien

Die Entwicklung objektorientierter Programmiermethoden und Sprachen wird uns auch in Zukunft Neuerungen bringen, die die Softwareentwicklung erleichtern, die die Kommunikation zwischen Mensch und Maschine verbessern und die dazu beitragen, dass Rechensysteme leistungsfähigere und sicherere Dienste für mehr Menschen anbieten können, als das heute der Fall ist. Dieser Abschnitt skizziert mögliche Entwicklungslinien im Zusammenhang mit objektorientierter Programmierung und Java.

**Verbesserungen.** Die existierenden objektorientierten Sprachen, Programmsysteme, Programmgerüste, Programmier- und Entwicklungsumgebungen bilden sicherlich nicht den Endpunkt der geschichtlichen und technischen Entwicklung. Dies kann man beispielsweise an der großen Zahl von Verbesserungen ablesen, die schon bis jetzt für die Java-Programmierungsumgebung und die Sprache Java selbst vorgeschlagen und angeboten wurden (parametrische Typen, Aufzählungstypen, etc.). Im Bereich der Programmierungsumgebung sind es zum einen leistungsfähigere Werkzeuge (schnellere Compiler, Just-in-time Compiler, bessere Debug-Unterstützung, Integration mit Entwurfswerkzeugen, etc.), zum anderen verbesserte oder neue Bibliotheken.

**Objektorientierung als konzeptioneller Ausgangspunkt.** Mit Hilfe der objektorientierten Programmierung konnte die Abstraktionsebene in der Software-Entwicklung angehoben werden. Dies wird besonders deutlich bei der Programmierung graphischer und verteilter Anwendungen, die gerade auch in Java von vielen (unnötigen) technischen Details befreit wurde. Die Objektorientierung bietet aber nicht nur Realisierungs- und Implementierungstechniken. Sie bildet auch den konzeptionellen Ausgangspunkt anderer Entwicklungen, die über die Programmierung weit hinausreichen. Am deutlichsten ist dies im Bereich objektorientierter Analyse-, Entwurfs- und Modellierungstechniken (vgl. Abschn. 1.4, S. 53). Ihre Konzepte haben aber auch andere, stärker technisch ausgerichtete Entwicklungen mit beeinflusst, z.B. die Realisierung komponentenbasierter Software. In diesem Zusammenhang spielen objektorientierte Programmgerüste eine wichtige Rolle, auch wenn Komponententechnologie sich nicht notwendig auf objektorientierte Programmiertechniken stützen müsste. Komponentenbasierte Programmgerüste verfolgen das Ziel,

*Software durch Anpassung und Komposition existierender Bausteine zu konstruieren. Zur Konstruktion werden zum Teil sogenannte Builder-Tools mit graphischer Unterstützung eingesetzt.*

Builder-Tools erlauben es i.A., Komponenten aus einer Bibliothek zu laden, mittels Drag-and-Drop-Techniken im Rahmen eines Fensters zu platzieren,

die Parameter und Eigenschaften der Komponenten mit graphischer Unterstützung einzustellen und zu verändern, Verbindungen zwischen Komponenten festzulegen sowie standardmäßige Ereignissteuerungen einzurichten. (Bereits ab Version 1.1 bietet Java mit den sogenannten Java Beans ein einfaches komponentenbasiertes Programmgerüst an; siehe z.B. [Eng97]. Ein auf Serverseite häufig verwendetes komponentenbasiertes Programmgerüst sind die Enterprise JavaBeans [DP00, GT00, EJB06].)

**Allgemeiner Ausblick.** Insgesamt gesehen bleiben noch viele Wünsche offen. Als Anregung kann die folgende naturgemäß unvollständige Wunschliste dienen:

- Bessere Unterstützung von persistenten Objekten; d.h. von Objekten, die eine Lebensdauer haben, die länger ist als die Lebensdauer des Programms, das sie erzeugt hat. Persistente Objekte sollten natürlich von laufenden Programmen importiert werden können. Auf diese Weise könnten insbesondere Dateisysteme und Datenbanken teilweise durch persistente Objekte abgelöst werden. Ein erster Schritt zur Unterstützung persistenter Objekte in Java wurde durch den Serialisierungsmechanismus getan; Serialisierung erhält aber nicht die Objektidentität (vgl. Unterabschn. 4.3.2.2).
- Bessere Unterstützung, um Objekte mit ihrem Programmcode einfach übers Netz verschicken zu können, ohne dass die Identität der Objekte dabei verloren geht. Ziel dabei ist es, eine programmtechnische Unterstützung zu schaffen, um leichter mobile Agenten zu realisieren. Mobile Agenten sind aktive, migrationsfähige Programme, die durch Kommunikation mit anderen Agenten selbständig komplexere Probleme lösen können (wie z.B. das Buchen der billigsten Flugverbindung unter bestimmten Vorgaben).
- Eine Verallgemeinerung von komponentenbasierten Programmentwicklungstechniken und bessere Unterstützungswerkzeuge für derartige Techniken. Ziel dabei sollte es sein, dass Endbenutzer ohne Spezialkenntnisse Softwarekomponenten problemlos zusammenbauen und konfigurieren können.
- Spezifikations- und Verifikationsunterstützung für objektorientierte Programme, mit der es im Rahmen der Qualitätsprüfung von Software relativ leicht möglich ist, aus der Korrektheit von Systemteilen auf die Korrektheit des Gesamtsystems zu schließen (vgl. [Mey92, MMPH99]).
- Eine adäquate Lösung für die Probleme in den Bereichen Zugriffsschutz und Nutzungsrechte. Dies hat insofern etwas mit Objektorientierung zu tun, als dass sich die Objektorientierung in offenen Netzen nur dann

allgemein verbreiten kann, wenn diese Probleme auch für die Objektebene gelöst werden können.

Dieser Ausblick geht bewusst über die objektorientierte Programmierung hinaus. Zentrale Aufgabe der Programmertechnologie bleibt es selbstverständlich in erster Linie, die Ziele derjenigen zu bedienen, die softwaretechnische Systeme realisieren wollen. Andererseits sind es häufig allgemeinere, übergeordnete Konzepte, die neue technologische Wege eröffnen. Dass objektorientierte Programmierung ein gutes Beispiel dafür bietet, braucht eigentlich nur noch erwähnt zu werden, um die von Abschn. 1.1 geöffnete Klammer zu schließen ;–)

## Selbsttestaufgaben

### Aufgabe 1: Simulation paralleler Clientzugriffe auf einen Fileserver (Sockets)

In dieser Aufgabe entwickeln wir einen einfachen Fileserver, der Textdateien übermittelt und mehrere Clients parallel bedienen kann, sowie eine Simulation mehrerer anfragender Clients. Der Server wird durch eine Klasse `FileServer` und die Clientanfragen durch eine Klasse `ParallelClients` implementiert.

a) Implementieren sie einen Server, der folgende Aufgaben erfüllt:

- Der Server wartet am Port 8181 auf eine Verbindung mit einem Client.
- Der Server kann beliebig viele Clients parallel bedienen.
- Der Server übermittelt die vom Client angeforderte Textdatei, die sich in demselben Verzeichnis wie der Server befinden soll.
- Wenn die Datenübertragung an einen Client gestartet und wenn die Datenübertragung beendet wird, gibt der Server eine entsprechende Meldung mit dem Namen des Clients aus. (Der Name des Clients wird dem Server bei der Dateianfrage mit übergeben).

b) Implementieren sie eine parallele Anfrage mehrerer Clients, die folgende Aufgaben erfüllt:

- Es werden  $n$  Clients gestartet, die alle dieselbe Textdatei vom Server anfordern (die Anzahl  $n$  wird vom Programm festgelegt).
- Die einzelnen Clients stellen die Socket-Verbindung mit dem Server her. Übergeben Sie hierzu die Serveradresse und den Dateinamen beim Programmstart als Parameter an die Main-Methode (`argv[0]` und `argv[1]`). Man kann das Programm mit den parallelen Clientanfragen also z.B. folgendermaßen starten: `java ParallelClients localhost Testdatei.txt`
- Jeder Client überträgt seine Nummer als Namen sowie den Namen der angeforderten Datei an den Server.
- Die Nummer des gestarteten Clients wird ausgegeben.
- Daten werden vom Server empfangen, aber nicht auf dem Bildschirm dargestellt.
- Sobald ein Client die gesamte Textdatei empfangen hat, wird eine entsprechende Meldung mit der Clientnummer ausgegeben.

Testen Sie die Klassen `FileServer` und `ParallelClients`!

Verwenden Sie als Serveradresse `localhost`, die immer den lokalen Rechner bezeichnet. Dazu müssen Sie das TCP/IP Protokoll auf Ihrem Rechner installiert haben und den Server und Client jeweils beide starten (unter Windows z.B. in zwei verschiedenen DOS-Eingabeaufforderungen und unter Linux in zwei verschiedenen Shells).

## Aufgabe 2: Client und Server für Systemzeit (RMI)

In dieser Aufgabe geht es um die Bereitstellung und das Auslesen der Systemzeit eines Servers. Auf Serverseite soll ein Zeitdienst realisiert werden, auf den von einem Client aus unter Verwendung von RMI zugegriffen werden kann. Entwickeln Sie dazu

- einen RMI-Zeitservice als Remote-Objekt, der das aktuelle Datum und die aktuelle Uhrzeit liefert;
- einen Prozess vom Typ `ZeitServer`, der eine Referenz auf das Remote-Objekt zur Verfügung stellt.

Dabei sind die Implementierungen so zu wählen, dass sie mit folgender Implementierung eines Clients korrekt zusammen arbeiten:

```
import java.rmi.*;

public class ZeitClient {
    public static void main(String[] args) throws Exception {

        Zeit uhr = (Zeit) java.rmi.Naming.lookup("rmi://localhost/Uhr");

        String aktuelleZeit = uhr.zeit();
        System.out.println(aktuelleZeit);
    }
}
```

*Hinweise:*

- `java.text.DateFormat.getDateTimeInstance().format(new java.util.Date())`  
erzeugt einen formatierten String mit Datums- und Zeitangaben. Mögliche Formate sind z.B. "30.03.2007 10:44:05" oder "Mar 30 2007 10:44:05 AM".
- Wird die Anwendung unter Windows getestet, müssen Namensserver, Serveranwendung und Client in verschiedenen DOS-Eingabeaufforderungen gestartet werden, unter Linux in verschiedenen Shells.

## Musterlösungen zu den Selbsttestaufgaben

### Aufgabe 1: Simulation paralleler Clientzugriffe auf einen Fileserver (Sockets)

- a) Die unten angegebene Klasse `FileServer` erfüllt zusammen mit den Klassen `DateiServer` und `BedienerThread` die Anforderungen aus der Aufgabenstellung. Die `main()`-Methode der Klasse `FileServer` erzeugt ein Objekt der Klasse `DateiServer`. Diese Klasse erzeugt für jede Clientanfrage einen neuen Thread, in dem die Datei an den Client übermittelt wird. Aus den vom Client gesendeten Daten wird dessen Name und der Name der angeforderten Datei ermittelt.

```
import java.io.*;
import java.net.*;
import java.util.*;

class FileServer {
    public static void main(String args[]) throws IOException {
        System.out.println("Starting FileServer...");
        DateiServer myServer = new DateiServer();
    }
}

class DateiServer {
    public DateiServer() throws IOException {
        ServerSocket serversocket = new ServerSocket (8181);
        while (true) {
            Socket socket = serversocket.accept();
            Thread bearbeiteAnfrage = new BedienerThread(socket);
            bearbeiteAnfrage.start();
        }
    }
}

class BedienerThread extends Thread {
    private Socket mySocket;

    public BedienerThread(Socket so) {
        mySocket = so;
    }

    public void run() {
        BufferedReader fromClient;
        BufferedWriter toClient;
        BufferedReader dateiLeser;
```

```

try {
    // Oeffne den Eingabestrom vom Client
    fromClient = new BufferedReader(new InputStreamReader
                                    (mySocket.getInputStream()));

    // Oeffne den Ausgabestrom zum Client
    toClient = new BufferedWriter(new OutputStreamWriter
                                   (mySocket.getOutputStream()));

    // Oeffnen der uebertragenen Datei
    String liesDatei = fromClient.readLine();
    StringTokenizer strToken = new StringTokenizer
        (liesDatei, " \r\n");

    // Clientnamen ermitteln
    String clientName = strToken.nextToken();
    String dateiName = strToken.nextToken();
    dateiLeser = new BufferedReader(new FileReader (dateiName));
    System.out.println("Bearbeite Anfrage Client " + clientName);

    int c = dateiLeser.read();
    while( c != -1) {
        toClient.write(c);
        c = dateiLeser.read();
    }
    dateiLeser.close();
    toClient.flush();

    System.out.println("Beende Anfrage Client " + clientName);
    mySocket.close();
} catch (Exception e) {
    System.out.println("Fehler beim Lesen der Datei");
}
}
}

```

- b) Es folgt die Implementierung einer Simulation von mehreren parallel anfragenden Clients. In der `main()`-Methode der Klasse `ParallelClients` werden nacheinander die Threads, welche die Client-Anfragen simulieren, gestartet. Dabei wird jedem Client ein Name zugewiesen, der dessen Ordnungszahl in der Reihenfolge der Aufrufe darstellt. Der jeweilige Thread liest Zeichen für Zeichen vom Server aus. Am Ende der Dateiübertragung gibt der Client eine Meldung aus, dass die Anfrage an den Server beendet ist.

```
import java.io.*;
import java.net.*;
import java.util.*;

class ParallelClients {

    public static void main(String args[]) throws IOException {
        System.out.println("Starting ParallelClients...");
        String serverName = args[0];
        String dateiName = args[1];
        int maxClients = 10;
        int clientZahl = 1;

        while (clientZahl <= maxClients) {
            Socket mySocket = new Socket(serverName, 8181);
            System.out.println("Starte Client " + clientZahl);
            Thread anforderer = new ClientThread
                (mySocket, dateiName, clientZahl);

            anforderer.start();
            clientZahl += 1;
        }
    }
}

class ClientThread extends Thread {

    private Socket neuSocket;
    private String datei;
    private int clientZahl;

    public ClientThread(Socket so, String str, int z) {
        neuSocket = so;
        datei = str;
        clientZahl = z;
    }

    public void run() {
        try {
            BufferedReader fromServer =
                new BufferedReader(new InputStreamReader
                    (neuSocket.getInputStream()));

            PrintWriter toServer =
                new PrintWriter(new OutputStreamWriter
                    (neuSocket.getOutputStream()));

            toServer.println(clientZahl + " " + datei);
            toServer.flush();
        }
    }
}
```



```

        int c = fromServer.read();
        while( c != -1) {
            c = fromServer.read();
        }
        System.out.println("Beende Client " + clientZahl);
        neuSocket.close();
    } catch (IOException e) {
        System.out.println("Fehler beim Lesen");
    }
}
}

```

Laufen Server- und Client-Anwendung auf einem gemeinsamen Rechner, müssen unter Windows Server-Anwendung und Client-Anwendung in verschiedenen DOS-Eingabeaufforderungen gestartet werden, unter Linux in verschiedenen Shells. In dem jeweiligen Fenster lässt sich durch die Textausgabe erkennen, welcher Client gestartet / beendet wurde und welche Clientanfrage bearbeitet wird.

### Aufgabe 2: Client und Server für Systemzeit (RMI)

Durch die Anweisung

`Zeit uhr=(Zeit)java.rmi.Naming.lookup("rmi://localhost/Uhr");` in der `main`-Methode von `ZeitClient` ist bereits der Name des Schnittstellentyps festgelegt (nämlich `Zeit`), der von dem Remote-Objekt zu implementieren ist. Dieser Typ muss ein Subtyp von `java.rmi.Remote` sein.

Durch die Anweisung

```
String aktuelleZeit = uhr.zeit();
```

wird impliziert, dass der Schnittstellentyp `Zeit` eine parameterlose Methode `zeit` enthält, deren Rückgabewert vom Typ `String` ist. Die Schnittstelle `Zeit` muss also wie folgt deklariert sein:

```
import java.rmi.*;
```

```
public interface Zeit extends Remote {
    String zeit() throws RemoteException;
}
```

Auf Serverseite muss es eine Klasse geben, die das Remote-Objekt und dessen Schnittstellentyp `Zeit` implementiert. Sie wird nach Java-Konventionen `ZeitImpl` genannt und muss die Klasse `UnicastRemoteObject` erweitern. Außerdem muss ihr Konstruktor eine `RemoteException` auslösen dürfen. Zur Formatierung der ausgelesenen Systemzeit wird die im Hinweis der Aufgabenstellung genannte Methode verwendet.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ZeitImpl extends UnicastRemoteObject
    implements Zeit {

    public ZeitImpl() throws RemoteException { }

    public String zeit() throws RemoteException {
        return java.text.DateFormat.getDateTimeInstance().
            format(new java.util.Date());
    }
}
```

Der Prozess, in dem sich das Remote-Objekt befindet, muss dieses mittels einer der statischen Methoden `bind` bzw. `rebind` der Klasse `java.rmi.Naming` unter dem Namen `Uhr` beim Namensserver anmelden. Der Name `Uhr` ist bereits durch den Parameter `rmi://localhost/Uhr` beim Aufruf der `lookup`-Methode im Client festgelegt.

```
public class ZeitServer {
    public static void main(String[] args) throws Exception {
        java.rmi.Naming.rebind("Uhr", new ZeitImpl());
    }
}
```

Bevor Server und Client gestartet werden, muss der Namensserver auf der Serverseite laufen. Dies kann durch Eingabe von `rmiregistry` erreicht werden (unter Windows in einer DOS-Eingabeaufforderung, unter Linux in einer Shell). Vor dem Start der Client-Anwendung muss der Server über `java ZeitServer` gestartet werden. Durch Aufruf von `java ZeitClient` wird der Client gestartet. Laufen Server und Client auf einem gemeinsamen Rechner, müssen unter Windows Namensserver, Server-Anwendung und Client in verschiedenen DOS-Eingabeaufforderungen gestartet werden, unter Linux in verschiedenen Shells.



# Literaturverzeichnis

- [AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
- [AG05] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 4. ed edition, 2005.
- [BA90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.
- [Bla94] Günther Blaschek. *Object-Oriented Programming with Prototypes*. Springer-Verlag, 1994.
- [BNS99] Ron Ben-Natan and Ori Sasson. *IBM San Francisco Developer's Guide*. McGraw-Hill, October 1999.
- [Bud01] T. Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, 2001.
- [COR02] CORBA Components Version 3.0, June 2002. Document: formal/02-06-65, Available at <http://www.omg.org/technology/documents/formal/components.htm>.
- [Cox86] Brad J. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.
- [DP00] S. Denninger and I. Peters. *Enterprise JavaBeans*. Addison-Wesley, 2000.
- [EJB06] JSR 220: Enterprise JavaBeans TM, Version 3.0, May 2 2006. Available at <http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html>.
- [Eng97] Robert Englander. *Developing Java Beans*. O'Reilly, 1997.
- [ES90] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

- [Far98] Jim Farley. *Java Distributed Computing*. O'Reilly, 1998.
- [Fla99] D. Flanagan. *Java in a Nutshell*. O'Reilly, Sebastopol, CA, 3rd edition, 1999.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.
- [GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Sun Microsystems Inc., 2nd edition, 2000.
- [GK96] Stephen J. Goldsack and Stuart J. H. Kent. *Formal Methods and Object Technology*. Springer-Verlag, 1996.
- [Goo99] Gerhard Goos. *Vorlesungen über Informatik*, volume 2. Springer-Verlag, 1999.
- [GR89] Adele Goldberg and David Robson. *SmallTalk 80: The Language*. Addison-Wesley, 1989.
- [GT00] V. Gruhn and A. Thiel. *Komponentenmodelle. DCOM, JavaBeans, Enterprise JavaBeans, CORBA*. Addison-Wesley, München, 2000.
- [HC97] Cay. S. Horstmann and Gary Cornell. *Core Java 1.1, Volume I — Fundamentals*. Prentice Hall, 1997.
- [HC98] Cay. S. Horstmann and Gary Cornell. *Core Java 1.1, Volume II — Advanced Features*. Prentice Hall, 1998.
- [Hoa74] C. A. R. Hoare. An operating system structuring concept. *Communications of the ACM*, 17:549–557, October 1974.
- [HS97] Brian Henderson-Sellers. *A Book of Object-Oriented Knowledge*. Prentice Hall, 1997.
- [JLHB88] E. Jul, H. M. Levy, N. C. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
- [Kee89] S. E. Keene. *Object-oriented Programming in COMMON LISP - A Programmer's Guide to CLOS*. Addison-Wesley, Reading MA, 1989.

- [Krü07] G. Krüger. *Handbuch der Java-Programmierung*. Addison Wesley, 6. auflage edition, 2007.
- [Kuh76] T. S. Kuhn. *Die Struktur wissenschaftlicher Revolutionen*. Suhrkamp, 1976.
- [Lam88] Günther Lamprecht. *SIMULA - Einführung in die Programmiersprache*. Vieweg Verlag, 3. neubearbeitete auflage edition, 1988.
- [Lea97] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1997.
- [Lip91] Stanley B. Lippman. *C++ Primer*. Addison-Wesley, 2nd edition, 1991.
- [LW94] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994. Kopiensammlung.
- [Mö8] H. Mössenböck. *Objektorientierte Programmierung in Oberon-2*. Springer-Verlag, 3. edition, 1998.
- [Mey92] B. Meyer. Design by contract. In D. Mandrioli and B. Meyer, editors, *Advances in object-oriented software engineering*. Prentice Hall, 1992.
- [Mey00] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2000.
- [MMPH99] P. Müller, J. Meyer, and A. Poetzsch-Heffter. Making executable interface specifications more expressive. In C. H. Cap, editor, *JIT '99, Java-Informations-Tage 1999*, Informatik aktuell. Springer-Verlag, 1999.
- [MMPN93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.
- [MPH98] P. Müller and A. Poetzsch-Heffter. Kapselung und Methodenbindung: Javas Designprobleme und ihre Korrektur. In C. H. Cap, editor, *JIT '98, Java-Informations-Tage 1998*, Informatik aktuell. Springer-Verlag, 1998.
- [NVP98] J. Noble, J. Vitek, and J. M. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98: Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

- [NW07] M. Naftalin and P. Wadler. *Java Generics and Collections*. O'Reilly, Sebastopol, CA, 2007. ISBN-10: 0-596-52775-6; ISBN-13: 978-0-596-52775-4.
- [PH00] A. Poetzsch-Heffter. *Konzepte objektorientierter Programmierung. Mit einer Einführung in Java*. Springer-Verlag, 2000. ISBN 3-540-66793-8.
- [Sie00] J. Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley & Sons, Inc., New York, 2000. ISBN: 0-471-29518-3.
- [Smo95] G. Smolka. The oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer-Verlag, 1995.
- [SOM94] C. Szypersky, S. Omohundro, and S. Murer. Engineering a programming language: The type and class system of Sather. In J. Gutknecht, editor, *Programming Languages and System Architectures*, *LNCS* 782, pages 208–227. Springer-Verlag, 1994.
- [Str00] B. Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley, Boston, 2000. ISBN: 0-201-70073-5.
- [Weg87] P. Wegner. Dimensions of object-based language design. In *Object Oriented Prog. Systems, Lang. and Applications*. ACM Press, 1987.

# Stichwortverzeichnis

- !=, ungleich, [29](#)
- ==, gleich, [29](#), [93](#), [480](#)
- A, [169](#), [229](#), [230](#), [489](#)
- A\_doch\_Null, [235](#)
- A\_nicht\_Null, [235](#)
- Abfangen von Ausnahmen, [38](#)
- abheben, [407](#)
- Ablaufverhalten von Threads, [385](#)
- ablegen, [411](#)
- abrupte Terminierung
  - der Ausführung, [38](#)
  - der Auswertung, [33](#)
- abstract, [230](#)
- Abstract Window Toolkit, [301](#), [307](#), [345](#), [349](#)
- abstrakte Klasse, [6](#), [166](#), [230](#), [274](#), [304](#), [310](#)
- abstrakte Methode, [230](#)
- abstrakter Datentyp, [260](#)
- abstraktes Modell, [301](#), [305](#), [307](#)
- Abstraktion, [145](#), [150](#), [156](#), [167](#), [216](#), [278](#)
- Ad-hoc-Polymorphie, [191](#)
- Ada95, [49](#), [103](#), [486](#)
- Adapterklasse, [272](#), [325](#)
- addMhtmlComponents, [360](#)
- Aktie, [202](#)
- aktive Komponente, [311](#)
- aktives Fenster, [309](#)
- aktueller Zustand, [75](#)
- Alias, [77](#), [240](#)
- Aliasing, [77](#)
- alle\_drucken, [22](#), [151](#), [152](#)
- Angestellte, [160](#), [162](#), [163](#), [165](#)
- anonyme Klasse, [192](#), [204](#), [328](#)
- Anweisung
  - bedingte, [35](#)
- Anweisungsblock, [34](#)
- Anwendung, [8](#), [303](#), [305](#), [306](#), [319](#), [333](#), [339](#), [347](#), [348](#), [350](#), [352](#), [361](#), [362](#)
- Anwendung, [235](#)
- Anwendung von Programmgerüsten, [346](#)
- Anwendungsnutzung, [234](#)
- Anwendungsschnittstelle, [347](#)
- Applet, [261](#), [310](#)
- Architektur, [263](#), [346](#)
- arraycopy, [92](#), [95](#)
- asynchrone Kommunikation, [436](#)
- atomare Aktion, [380](#)
- Attribut, [20](#), [44](#), [75](#), [78](#)
  - statisches, [93](#)
  - unveränderliches, [89](#)
- Attributdeklaration, [75](#)
- Attributzugriff, [81](#)
- Auffinden der Schlüsselabstraktionen, [131](#)
- Aufgabenbereich, [19](#), [131](#), [302](#)–[304](#)
- Auflösen von Überladung, [91](#)
- Auftrag, [242](#), [246](#)
- Auftragsverwaltung, [242](#), [246](#)
- Aufzählungstyp
  - als eigener Typ in Java 5.0, [194](#)
  - als Schnittstelle, [159](#)
  - am Beispiel Status, [242](#)
- Ausdruck, [32](#)
- Ausdrücke als Anweisung, [34](#)
- Ausführung, [34](#), [34](#)
  - abrupt terminierende, [38](#)
  - normal terminierende, [38](#)
- Ausführungssemantik, [40](#), [238](#)



- Ausführungsstrang, 262, 379, 380, 442, 457
- Ausnahme, 39, 263
  - abfangen, 38
- Ausnahmebehandlung, 38, 39, 41, 171, 257, 258, 263, 265, 310, 463, 466
- Ausnahmesituation, 38
- Auswertung
  - abrupt terminierende, 33
  - nicht-strikte, 33
  - normal terminierende, 33
  - strikte, 33
- Auswertung von Ausdrücken, 33
- Autoboxing, 172, 173
  - Boxing, 172
  - Unboxing, 172
- AWT, 301, 307, 345, 349
- AWT-Komponente, 309, 310
  
- B, 169, 229, 489
- Bank, 398, 407, 407
- BankTest, 407
- BaseFrame, 327, 329, 334
- BaseFrameTest, 327
- Basisdatentypen, 24, 27, 94, 157, 171, 440, 460
- BathRoom, 115
- Baustein, 257, 272
- BedienFenster, 359, 457
- bedingte Anweisung, 35
- Behaelter, 151, 167
- Behälter, 116
- Behälter-Datentyp, 99
- Behälter-Komponente, 309
- Behandeln von Ereignissen, 325
- benannte Konstante, 89
- benutzen
  - Klasse benutzt Klasse, 132
- Benutzungsbeziehung, 132, 132
- Beobachter, 201, 318
- Beobachter registrieren, 316
- Beobachtermuster, 201, 316
- beschränkt parametrische Polymorphie, 190
- beschränkt parametrische Typen, 180
- BETA, 49, 51, 132, 490–492
- Betreten eines Monitors, 401
- Beziehungen zwischen Klassen, 131
- Bibliothek, 265
- bind, 472
- Block, 34, 113
  - synchronisierter, 401, 403
- blockierter Thread, 385
- Blockschachtelung, 113
- Boersianer1, 203, 204
- Boersianer2, 203, 204
- boolean, 27
- BorderLayoutTest, 335
- Boxing, 172
- break-Anweisung, 37
- BrMark, 343, 358
- browse, 124
- browse.util, 124, 129
- Browser, 85, 261, 306, 350, 447
- Browser, 86, 95, 97, 99, 224
- Browseroberfläche, 351
- BufferedReader, 274
- ButtonTest, 329
- byte, 27
  
- C, 88, 169, 229, 231
- C++, 49, 51
- callback-Mechanismus, 192
- Canvas, 333
- CanvasTest, 334
- case, 37
- Cast, 33
- catch-Klausel, 38
- char, 27
- charAt, 93
- CharEingabeStrom, 268
- Class, 279
- class, 78, 160
- Class-Objekt, 401
- ClassCastException, 187
- Client, 446

- Client-Server-Architektur, [437](#)
- CLOS, [49](#), [51](#), [486](#)
- close, [273](#)
- ClosingBeobachter, [327](#)
- Color, [153](#)
- Component, [310](#)
- Container, [310](#)
- container, [99](#)
- Controller, [347](#)
- CORBA, [263](#), [441](#), [469](#)
- CORBA-Komponentenmodell, [441](#)
- CountServiceImpl, [476](#)
- D, [229](#), [231](#)
- Darstellung, [305](#), [347](#), [361](#)
- Darstellungselement, [308](#)
- DataInput, [276](#)
- DataOutput, [276](#)
- DateiClient, [447](#)
- DateiLeser, [272](#)
- DateiServer, [445](#)
- DateiZugriff, [275](#)
- DateiZugriffTest, [275](#)
- Datentyp
  - abstrakter, [260](#)
- default access, [129](#), [233](#)
- default-Fall, [37](#)
- default-Konstruktor, [42](#), [80](#), [217](#), [224](#)
- Deklaration von Subtyping, [159](#)
- deklarative Programmierung, [15](#), [17](#)
- Dialog, [306](#), [347](#), [352](#)
- Dialog, [337](#)
- Dialogfenster, [310](#), [337](#), [339](#), [350](#), [352](#)
- DiningRoom, [115](#)
- direkter Subtyp, [159](#)
- direkter Supertyp, [159](#), [161](#)
- dispose, [325](#)
- DNS-Protokoll, [448](#)
- do-Anweisung, [35](#)
- DoButton, [329](#)
- DoButton2, [331](#)
- DoButton23Test, [331](#)
- DoButton3, [331](#)
- DoButtonTest, [329](#)
- Domänenkenntnis, [303](#)
- doppeltverkettete Liste, [99](#)
- double, [27](#)
- Druckbar, [150](#), [152](#), [155](#), [158](#), [165](#), [166](#)
- drucken, [43](#), [43](#), [151](#)
- DruckMitLaufNr, [167](#)
- dynamisch, [46](#)
- dynamische Bindung, [46](#), [93](#), [152](#), [168](#), [226](#)
- dynamische Methodenauswahl, [145](#), [152](#), [168](#), [174](#), [238](#), [485](#)
- dynamisches Binden, [24](#)
- dynamisches Laden, [10](#), [52](#), [83](#), [463](#), [474](#)
- EA-blockiert, [386](#)
- echte Polymorphie, [191](#)
- Eiffel, [49](#), [51](#), [54](#), [103](#), [490](#)
- Eigenschaften von Objekten, [75](#)
- eigenständiger Baustein, [259](#)
- EinBroker, [478](#)
- EinClient, [478](#)
- EinServer, [476](#)
- einstellenLetzten-  
Hintergrund, [154](#)
- EinTest, [480](#)
- EinTyp, [480](#)
- einzahlen, [406](#)
- elementare Komponente, [309](#), [328](#)
- elementares Ereignis, [314](#)
- Empfängerobjekt, [5](#), [81](#), [377](#)
- enabled component, [311](#)
- eng kooperierende Bausteine, [259](#)
- entfernter Methodenaufruf, [443](#), [458](#), [463](#), [466](#), [470](#)
- Entry, [100](#), [101](#), [103](#), [114](#), [221](#)
- Entry<ET>, [104](#)
- Entwerfen von Klassen, [84](#)
- Entwickeln von Layout-Managern, [343](#)
- Entwicklungsmethodik, [346](#), [349](#)
- Entwurfsmuster, [201](#)
- enum, [194](#)
- equals, [93](#), [175](#), [480](#)

- Ereignis, 308, 314, 317, 319, 332
  - behandeln, 325
  - elementares, 314
  - semantisches, 314
  - tritt an Komponente auf, 314
- Ereignisquelle, 314
- Ereignissorte, 314
- Erreichbarkeitsbeziehung, 132
- erreichen
  - Objekt erreicht Objekt, 132
- Error, 264
- Erzeuger, 410, 467, 473
- ErzeugerVerbraucherTest, 412
- evaluation, 33
- Event, 308
- Exception, 265
- Exceptions, 174
- execution, 34
- expression, 32
- ExtendedList, 221, 222
- extends, 155, 160
- ExtListIterator, 222
- fairen Scheduling, 380, 396
- Fairness, 400
- Farben, 159, 194
- Fehler, 264
- Feldelement, 29
- Fenster, 308, 309
  - aktives, 309
  - inaktives, 309
- FensterBeobachter, 326
- FensterBeobachterTest, 326
- FileReader, 272, 275
- FilterMain, 200
- FilterPraedikat, 199
- final, 89, 94, 97, 117, 159, 236, 242, 331, 338, 344
- finally-Klausel, 38
- Firma, 166
- FirstFloor, 115
- float, 27
- FlowLayoutTest, 340
- flush, 446
- Fokus, 309, 314
- for-Anweisung, 35
- formaler Parameter, 44
- Frame, 153, 310, 324
- FrameTest, 324
- Framework, 152, 303
- FTP-Protokoll, 448
- Funktion höherer Ordnung, 15
- funktionale Programmierung, 15
- Funktionalität, 8, 10, 118, 155, 161, 238, 261, 272, 278, 302, 350, 470
- garbage collection, 488
- Geheimnisprinzip, 111
- gemeinsamer Speicher, 391, 404
- GenauB, 232
- Gerätesteuerung in einem Haushalt, 19
- Gerüst, 301, 303, 303, 305, 307, 346
- geschützt, 234, 235
- geschützte Attribute, 220
- getBackground, 153
- getEinstellungsdatum, 160
- getFile, 453
- getGehaltsklasse, 160
- getHost, 453
- getInhalt, 76, 79, 112
- getMinimumSize, 341
- getPreferredSize, 341
- getProtocol, 453
- getSource, 316
- getTitel, 76, 79, 112
- Graphikkontext, 333
- GrossBuchstabenFilter, 270
- GUI, 305
- hat-ein-Beziehung, 132
- hat\_geburtstag, 43, 44, 44, 219
- Hauptfenster, 310, 323, 325, 327
- heterogenes System, 440
- HinweisFenster, 339
- HinweisFensterTest, 338
- holenSeite, 117
- homogenes System, 439

- HrefButton, 361
- HTML, 86, 112, 350, 455
- HTTP-Protokoll, 448
- HTTPS-Protokoll, 448
- Hypertext Markup Language, 86
- Hypertext-Referenz, 350, 358
- Identität, 18, 25, 461, 480, 495
- if-Anweisung, 35
- imperative Programmierung, 13
- implements, 160
- impliziter Parameter, 79, 82, 93, 220, 493
- import-Anweisung, 124, 128
- in, 94
- inaktive Komponente, 311
- inaktives Fenster, 309
- indexOf, 92, 105, 112
- Information Hiding, 111
- inheritance, 217
- inhomoge Liste, 189
- initialer Zustand, 13
- initialisieren, 97
- Initialisierungsblock, 89
  - nicht-statisch, 89
  - statisch, 89
- innere Klasse, 113, 114–116, 118, 127, 131
- InputStream, 273
- instanceof-Operator, 187, 344, 395
- Instanz, 78, 80, 205
- Instanziiieren, 78, 103, 105
- Instanzvariable, 78, 93, 391, 398, 407
- Int, 172
- int, 27
- Integer, 38
- interaktive Steuerung, 85, 95
- interaktiveSteuerung, 98, 225, 226
- interface, 158
- interrupt, 392
- IntPair, 237
- IntTriple, 237
- IRC-Protokoll, 448
- is-a relation, 149
- ist-ein-Beziehung, 133, 149, 155, 156
- Iterable, 193
- Iterator, 193
- Java, VIII
  - java, Java-Bytecode-Interpreter, 47
  - Java-Bibliothek, 261
  - java.applet, 261
  - java.awt, 261, 322
  - java.awt.event, 322
  - java.beans, 262
  - java.io, 262, 272, 273
  - java.lang, 128, 262
  - java.math, 262
  - java.net, 262, 443, 453
  - java.rmi, 262, 470
  - java.security, 262
  - java.sql, 262
  - java.text, 262
  - java.util, 262
  - javac, Java-Compiler, 47
  - javax.accessibility, 263
  - javax.swing, 263
- K, 402
- KaffeeMaschine, 266
- Kapselung, 51, 73, 111, 233
- KeinKaffeeException, 265
- Klasse, 5, 51, 73, 75, 78, 100, 102, 103, 124
  - abstrakte, 230
  - anonyme, 204
  - benutzt Klasse, 132
  - entwerfen, 84
  - generische, 103
  - innere, 113
  - lokale, 204
  - parametrische, 100, 103
  - rekursive, 99
- Klassen als Objekte, 492
- Klassenattribut, 88, 93, 96, 132
- Klassendeklaration, 78
- Klassenentwurf, 132

- Klasseninvarianten, 111
- Klassenkonzept, 73, 74, 260, 485
- Klassenmethode, 82, 88, 93, 226
- Klassentyp, 78, 81, 128, 157, 161, 162, 166, 169
- Klassifikation, 5, 10, 21, 147, 148, 156, 162, 167, 169, 310, 484
- Klonen von Objekten, 25, 74, 75, 493
- Kommentarklammern, 30
- Kommentarzeichen, 30
- Kommunikation
  - asynchron, 436
  - Organisation von, 437
  - synchrone, 436
- Kommunikationsart, 49, 436
- Kommunikationsmittel, 437
- Komponente
  - aktive, 311
  - einer Klasse, 78
  - einer Oberfläche, 308
  - eines Software-Systems, 83, 238
  - inaktive, 311
  - sichtbare, 311
- Komponentendarstellung, 332
- Komponentenschnittstelle, 238
- Komponententyp, 28, 29
- Komponententyp des AWT, 309, 311
- Konflikt, 127, 129
- konformes Verhalten, 175, 485
- Konkatenation von Zeichenreihen, 85
- Konsole, 96
- Konstante, 27, 30, 32, 339
  - benannte, 89
- Konstruktor, 43, 77, 80, 96, 217, 226, 227
- Konstruktoraufruf, 80
- Konstruktordeklaration, 80
- KonstruktorProblemTest, 227
- Konstruktorsignatur, 80
- Kontravarianz, 175
- Kontrollstruktur, 35, 493
- kooperierende Threads, 410
- Kopieren von Objekten, 25
- Koroutine, 491
- Kovarianz, 175
- kritischer Bereich, 400, 406
- LabelCanvas, 334
- LabelTest, 328
- lauffähiger Thread, 385
- Laufzeitkeller, 238, 264, 391
- Layout-Manager, 152, 313, 322, 329, 335, 336, 340, 341, 343
  - entwickeln, 343
- Lebendigkeitseigenschaft, 397
- Lebewesen, 389
- Leerwesen, 389
- length, 93
- letzterHintergrund, 153
- Lexik, 200
- LinkedList, 99, 101, 102, 103, 111, 114, 117–119, 121, 130, 189, 193, 220, 221, 248, 258, 279
- LinkedList<ET>, 104
- List, 487
- ListElem, 117
- Listener, 316
- ListIterator, 119, 121, 130, 221
- listIterator, 118
- localhost, 446
- logische Programmierung, 16
- lokale Klasse, 201, 204
- lokale Parallelität, 378, 382, 416
- long, 27
- Main, 46, 47
- Mehrfachvererbung, 51, 229, 231, 488
- MeinThread1, 383
- MeinThread2, 383
- MemoFrame, 154
- Methode, 5, 20, 23, 42, 44, 75, 76, 78, 81, 93
  - abstrakte, 230
  - höherer Ordnung, 492
  - statische, 82, 93
  - synchronisierte, 401
- Methodenaufruf, 81

- Methodendeklaration, 78
- Methodenrumpf, 38, 78
- Methodensignatur, 78, 100, 160, 260, 267
- MHTML, 350
- modaler Dialog, 339
- Model-Schnittstelle, 347
- Model-View-Controller-Architektur, 347
- Modell
  - abstraktes, 305
- Modellierung der realen Welt, 19
- Modifikatorenliste, 78
- Modula-3, 49, 486
- Modularisierung von Programmen, 123
- Monitor, 386, 393, 401, 402, 406
- Monitor blockiert, 385, 402, 406
- Multi-Methoden, 493
- multiple inheritance, 231
- MVC-Architektur, 346, 347, 349, 362
- MyClassIsMyCastle, 115
- Nachricht, 5, 25, 34, 42, 74, 75, 171, 308, 377, 437, 484, 485, 492
  - verschicken, 23
- Name
  - vollständiger, 126
- namen, 127
- Namensraum, 123
- native, 95
- NetzSurfer, 357
- NetzSurfer2, 457
- neuer Thread, 385
- new, 80
- nextIndex, 118
- nicht-strikte Auswertung, 33
- Nichtdeterminismus, 396, 397
- NichtDruckbar, 173
- NNTP-Protokoll, 448
- nocheinpaket, 126
- normale Terminierung
  - der Ausführung, 38
  - der Auswertung, 33
- NoSuchElementException, 111
- notify, 386, 401, 405
- notifyAll, 386, 405
- null, 28
- null-Referenz, 28, 264
- NumberFormatException, 38
- Oberflächenkomponente, 8, 10, 148, 304, 307, 308, 319
- Oberflaeche, 395
- Oberklasse, 227
- Object, 149, 159, 169, 187, 189, 262
- Object Management Group, 440
- Object Pascal, 49
- ObjectInput, 276
- Objective C, 49
- ObjectOutput, 276
- Objekt, 4, 10, 24, 25, 30, 42, 49, 51, 73, 78, 147, 156
  - bearbeitet Nachricht, 18
  - der realen Welt, 19
  - einer Klasse, 78
  - erreicht Objekt, 132
  - hat einen Zustand, 18
  - vs. Wert, 25
- objektbasiert, 74, 77
- Objektbeschreibung, 51
- Objekterzeugung, 30, 80, 472
- Objektgeflecht, 31, 131, 238, 240, 265, 268, 279, 281, 349, 463, 480, 481
- Objektkonzept, 484
- objektlokale Variable, 42, 75, 78, 93
- objektorientierte Programme, 82
- objektorientierte Programmierung, 3, 4, 6, 18, 42, 54, 74, 131, 302, 435, 483
- objektorientierter Entwurf, 53, 131
- objektorientiertes Grundmodell, 19, 20, 21, 377
- Objektreferenz, 26, 30, 278, 460
- Objektströme, 279
- Objekttyp, 28, 157, 490
- Objektverhalten, 75
- öffentlich, 128, 129

- OMG, 440
- Operationen in Java, 29
- Organisation der Kommunikation, 437
- out, 94
- OutputStream, 273, 279
- package, 124
- Paket, 123, 124, 128, 233, 442
- paketlokal, 129
- paketlokaler Zugriff, 233, 234
- paketlokales Programmelement, 129
- Panel, 310, 335
- Paradigmen der Programmierung, 11
- Parallelität, 377
  - lokale, 378
- parametrische Klassen, 103
- parametrische Polymorphie, 190
- Parametrisierung von Klassen, 100
- parseInt, 38
- partielle Ordnung, 157
- Pattern, 491
- Persistenz, 278, 495
- Person, 43, 48, 158, 165, 167, 218
- Person-Objekt, 42, 43, 46
- Polymorphie, 145, 169, 189, 191
- POP3-Protokoll, 448
- Port, 442
- PrefixAoderB, 200
- print, 94
- PrintServiceImpl, 476
- PrintWriter, 275, 446
- Prioritäten von Threads, 386
- privat, 112
- private, 112
- privates Programmelement, 112, 114
- PrivateTest, 237
- Produkt, 412, 460, 464, 469
- Programmausführung, 18, 74, 83, 89, 95, 377, 379, 414
- Programmbaustein, 9, 257
- Programmelement, 129
  - geschütztes, 234
  - öffentliches, 129
  - paketlokales, 129
  - privates, 112, 114
- Programmgerüst, 53, 152, 257, 259, 301, 302, 303, 304, 322, 346
  - Anwendung von, 346
- Programmierung, 4
  - deklarative, 15
  - funktionale, 15
  - für Vererbung, 224
  - imperative, 13
  - logische, 16
  - objektorientierte, 18
  - prozedurale, 12
- Programmstruktur, 20, 224, 484
- protected, 234
- Protokoll, 259, 440, 444, 449, 452, 455
- Prototyp-Konzept, 74
- prototypbasiert, 51, 493
- Prozedur, 13
- prozedurale Programmierung, 12
- Prozedurzeiger, 199
- public, 129
- Quelle eines Ereignisses, 314
- RandomAccessFile, 276
  - read, 268, 273
  - readString, 96
- reale Parallelität, 378
- reale Welt, 19
- rebind, 472
- rechenbereit, 402
- rechenbereiter Thread, 385
- rechnender Thread, 385
- Referenz, 26
- Referenztyp, 157, 171, 403
- Reflexion, 51
- Registrieren von Beobachtern, 316
- rekursiv, 99, 307, 335, 487
- rekursive Abhängigkeit, 99, 259, 323
- rekursive Klassendefinitionen, 73
- rekursive Klassendeklaration, 99
- Remote, 470
- Remote-Objekt, 468, 468, 470, 472

- RemoteTyp, [480](#)
- Repräsentation, [239](#)
- Request-Broker-Architektur, [438](#), [441](#)
- return-Anweisung, [38](#), [102](#)
- RingPuffer, [413](#), [470](#)
- RingPufferImpl, [471](#)
- RingPufferServer, [464](#), [471](#)
- RMI, [433](#), [458](#), [474](#)
- RMI-Registry, [448](#)
- rmiregistry, [472](#)
- Rumpf
  - einer Klasse, [75](#)
  - einer Methode, [79](#)
  - eines Konstruktors, [80](#)
- run, [383](#)
- Runnable, [382](#)
- RuntimeException, [264](#)
- Scheduler, [380](#), [396](#), [398](#), [415](#)
- Scheduling
  - nicht-preemptives, [380](#)
  - preemptives, [380](#)
- Scheduling von Threads, [396](#)
- schlafend, [386](#), [393](#)
- Schleifen-Anweisung, [35](#)
- Schlüsselabstraktion, [131](#)
- Schnittstellenbildung, [21](#), [112](#), [156](#), [484](#), [485](#)
- Schnittstellendeklaration, [124](#), [157](#), [158](#)
- Schnittstellentyp, [157](#), [159](#), [161](#), [162](#), [164](#), [166](#), [191](#), [470](#)
- schützenswerte Referenz, [240](#)
- ScrollPane, [310](#)
- SecondFloor, [115](#)
- Self, [51](#), [493](#)
- self-Objekt, [44](#), [79](#)
- semantisches Ereignis, [314](#)
- Senderobjekt, [5](#), [49](#), [377](#)
- sequentielle Konsistenz, [391](#), [404](#)
- Serializable, [281](#), [464](#)
- Server, [443](#), [457](#)
- ServerSocket, [443](#)
- Service, [477](#)
- ServiceLookup, [477](#)
- ServiceLookupBImpl, [476](#)
- ServiceLoopupImpl, [476](#)
- ServiceThread, [458](#), [465](#)
- setBackground, [153](#)
- setLocation, [324](#)
- setSize, [324](#)
- setVisible, [325](#)
- short, [27](#)
- Sicherheitseigenschaft, [397](#)
- sichtbare Komponente, [311](#)
- Signatur
  - einer Methode, [78](#)
  - eines Konstruktors, [80](#)
- SILC-Protokoll, [448](#)
- SimpleHTTPClient, [453](#)
- Simula, [6](#), [49](#), [490](#), [492](#)
- SIP-Protokoll, [448](#)
- Skeleton-Objekt, [468](#), [469](#)
- Skript, [150](#)
- sleep, [386](#), [393](#)
- slogan, [127](#)
- Smalltalk, [7](#), [26](#), [49](#), [51](#), [261](#), [347](#), [492](#), [493](#)
- SMTP-Protokoll, [448](#)
- Socket, [442](#)
- Socket, [444](#)
- Software-Architektur, [346](#)
- Software-Komponenten, [238](#)
- Speicher
  - gemeinsamer, [391](#), [404](#)
  - zentraler, [391](#)
- Speicherbereinigung, [488](#)
- Speichermodell, [391](#), [404](#)
- Spezialisierung, [6](#), [10](#), [24](#), [44](#), [51](#), [145](#), [152](#), [156](#), [167](#), [215](#), [216](#), [220](#), [222](#), [226](#), [237](#), [350](#), [443](#), [485](#)
- SSH-Protokoll, [448](#)
- Standard-Paket, [128](#)
- start, [96](#), [224](#), [383](#), [385](#)
- starvation, [396](#)
- static, [93](#), [116](#)
- statisch, [46](#)



- statische Methode, 93
- statisches Attribut, 93
- statisches Binden, 24
- Status, 242, 246
- Steuerung, 305, 306, 308, 314, 347, 350, 361, 362
- strikte Auswertung, 33
- String, 88, 91, 94, 262
- String-Objekt, 38
- StringBuffer, 262, 279
- StringLeser, 269
- StringTokenizer, 341
- Ströme, 268
- Strukturieren von Klassen, 110, 113
- Stub-Objekt, 468
- Stub-Skeleton-Mechanismus, 468
- Student, 44, 150, 152, 161, 165, 218
- Student-Objekt, 42, 46
- Subclassing, 215, 228, 233
- Subklasse, 218
- Subtyp, 45, 155–158, 159, 165, 169, 173, 267, 281
  - direkter, 159
- Subtyp, 174
- Subtyp-Beziehung, 159
- Subtyp-Ordnung, 157
- Subtyp-Polymorphie, 189
- Subtyping, 24, 45, 157, 157, 168, 228, 485, 490
- Subtyping-Grundregel, 157
- super
  - Aufruf der überschriebenen Methode, 220
  - Aufruf des Superklassenkonstruktors, 217
- Superklasse, 160
- Superklasse, 218
- Supertyp, 159, 164, 166, 173–175
  - direkter, 159
- Supertyp, 174
- switch-Anweisung, 37
- synchrone Kommunikation, 436
- Synchronisation, 397
  - mit Monitoren, 406
- synchronisierte Methode, 401
- synchronisierter Block, 403
- synchronized, 401, 403
- syntaktische Bedingung von Subtyping, 173
- System, 88, 94, 112
- System von Klassen, 302, 303, 322
- TCP/IP, 446
- Teil-von-Beziehung, 132
- Telnet-Protokoll, 448
- Test, 105
- Test1, 82
- Test2, 82
- TestIntWrapper, 172
- TestMemoFrame, 154
- testpaket, 126
- TextFenster, 85
- TextFieldTest, 332
- TextLayout, 344
- TextLayoutTest, 345
- this-Objekt, 44, 79, 118, 220
- Thread, 233, 377, 382, 382, 386, 388, 392, 393, 398, 400, 407, 458, 464
  - Ablaufverhalten, 385
  - blockierter, 385
  - lauffähiger, 385
  - neuer, 385
  - rechenbereiter, 385
  - rechnend, 385
  - schlafender, 386
  - toter, 385
  - wartender, 385
  - Zustandsverhalten, 385
- Thread, 382, 386, 392
- Thread-Prioritäten, 386
- throw-Anweisung, 38, 41, 266
- Throwable, 264, 267
- throws, 267
- throws-Deklaration, 79
- Toolkit, 53, 301, 307
- toter Thread, 385
- transfer, 398, 406

- try-Anweisung, 38, 40, 41, 79, 267
- Typ, 26
  - in Java, 28
  - parametrischer
    - beschränkt parametrischer, 180
    - Platzhalter, 185
- Typ-Platzhalter, 185
- Typdeklaration, 124, 128, 150, 152, 165, 166, 323
- Typisierung, 156
- Typkonstruktor, 28
- Typkonvertierung, 33, 186, 187, 189, 268, 489
- Typkonzept, 14, 100
- Typsystem, 493
- Typtest, 187
  
- Überladen, 88, 91, 92, 96, 191
- Überladung
  - auflösen, 91
- Überschreiben, 155
- Überschreiben, 45, 215, 220, 234–236
- Übersetzungseinheit, 124, 124, 126, 128
- UmlautSzFilter, 271
- unabhängiger Baustein, 258
- Unboxing, 172
- UnicastRemoteObject, 472
- Unicode-Zeichen, 27, 274
- Unterklasse, 227
- unveränderliche Klassen, 235, 236
- unveränderliches Attribut, 89
- URL, 433, 452, 455, 460, 472
- URL, 457
- uses relation, 132
  
- valid, 311
- valueOf, 94
- Variable
  - lokale, 77
  - objektlokale, 42, 75, 78
- Variablen, 12, 26, 28
- Vererbung, 5, 6, 44, 51, 73, 147, 153, 156, 160, 166, 217, 218, 220, 222, 227, 228, 233, 234, 236, 302, 308, 484, 490
  - im engeren Sinne, 217
- Vererbungskonzept, 485
- Vererbungsnutzung, 234
- Vererbungsschnittstelle, 227
- VererbungsTest, 218
- Verhalten von Objekten, 75
- Verhungern, 396
- Verklemmung, 397
- Verschicken einer Nachricht, 23
- verteilte Anwendung, 9, 50, 53
- verteilte Objekte, 458
- verteilt System, 435
  - heterogenes, 440
  - homogenes, 439
- VerwAng, 150
- VerwAngestellte, 163, 165
- View, 347
- virtuelle Parallelität, 378
- virtuelle Welt, 19
- visible, 311
- visible, 325
- void, 79
- volatile, 392, 404
- vollständiger Name, 126
- VorZukBrowser, 223, 226
  
- W3Seite, 76, 80, 111–113, 129
- W3SeitenTest, 76
- W3Server, 85, 117, 120
- wait, 401
- wait/notify-Mechanismus, 404
- wartend, 393
- wartender Thread, 385
- WButton, 393, 395
- Welt, 390
- Wert, 25
  - in Java, 27
- Wesen, 389, 394
- while-Anweisung, 35
- Wiederverwendung, 254
- Window, 310
- WindowAdapter, 327

WissAng, 150  
WissAngestellte, 163, 165  
WordComponent, 342, 358  
World Wide Web, 75  
Wrapper-Klasse, 172, 262  
writeString, 96  
WWW, 75  
  
Zeichenreihen  
    konkatenieren, 85  
Zeiger, 26, 487, 489, 490  
zentraler Speicher, 391  
Zugriffsmodifikator, 112  
Zugriffsrecht  
    geschütztes, 234  
    öffentliches, 129  
    paketlokales, 129  
    privates, 115  
zusammengesetzte Namen, 115  
Zustand, 12, 19, 25, 26, 42, 260, 278,  
    460  
    aktueller, 75  
    eines Objekts, 75  
Zustandsverhalten von Threads, 385  
Zuweisung, 13, 29, 34, 489