Vorwort

Vielen Dank, lieber Leser[1], dass Sie sich Zeit für dieses Vorwort nehmen. Ich werde mich kurz fassen.

Programmiersprachen lassen sich nach vielen Kriterien klassifizieren. Eine nicht ganz ernst gemeinte Einteilung unterscheidet *blue collar languages* und *white collar languages*. Erstere benutzen Leute in Blaumännern und Letztere Leute mit weißen Hemden. Die Leute in Blaumännern erledigen Arbeit, die Leute in weißen Hemden denken sich Arbeiten aus.

Java fällt in die erste Kategorie. In diesem Sinn zielt dieses Buch auf den Einsatz von Java zum Lösen praktischer Probleme. Dabei geht es zwar vordergründig um die konkrete Programmierung, hinter den einzelnen Themen stehen aber auch konzeptionelle Fragen der Softwareentwicklung. Das Buch ist keine Sammlung von Rezepten oder gar ein Java-Referenzwerk, sondern eine Darstellung praktischer Lösungsansätze mit Java vor dem Hintergrund der Informatik.

Beim Leser werden Kenntnisse der Java-Programmierung vorausgesetzt. Eine kompakte Zusammenstellung dieser Voraussetzungen finden Sie im Anhang A.

Inhalt

Jedes Kapitel dieses Buches greift ein abgegrenztes Thema auf und entwickelt Bausteine und Leitlinien für Java-Programme mit den betreffenden Sprachmitteln. Die folgende Liste gibt einen Überblick: Kapitelübersicht

1. **Ein- und Ausgabe:** Ausgehend von der Ein- und Ausgabe über den Bildschirm und die Tastatur beschäftigt sich dieses Kapitel mit der Verarbeitung von Dateien. Den Abschluss machen programmgesteuerte Datei-Operationen im Filesystem.
2. **Serialisierung:** Serialisierung erlaubt es, Java-Datenstrukturen im Filesystem zu sichern und wieder zu rekonstruieren. Damit können Objekte über Programmstarts hinweg erhalten oder zwischen verschiedenen Java-Programmen ausgetauscht werden.
3. **XML:** XML ist als Datenformat heute nicht mehr wegzudenken. Java unterstützt den Umgang mit XML-Dokumenten mit vielen Bibliotheksklassen und

[1] Im gesamten Buch stehen syntaktisch männliche Personenbezeichnungen, wie zum Beispiel „Leser“, für inhaltlich neutrale Tätigkeiten und Rollen. Solche Bezeichnungen beziehen sich immer auf Personen beiderlei Geschlechts.

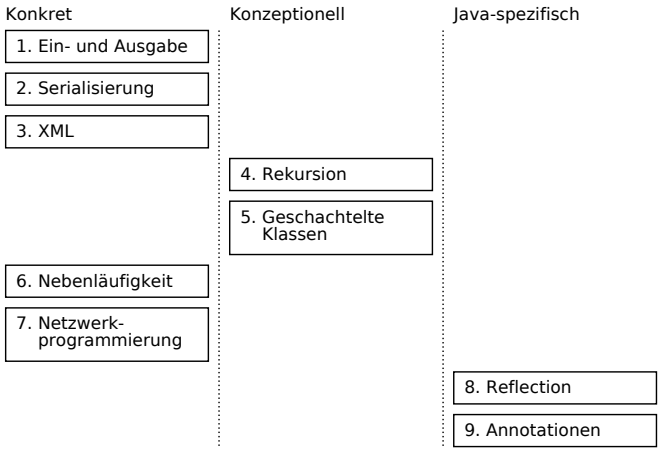
-methoden. Dieses Kapitel zeigt, wie ein Programm XML-Dokumente lesen, prüfen, umbauen und neu erzeugen kann.

4. **Rekursion:** Rekursion beruht auf Methoden, die sich selbst aufrufen. So fremdartig diese Idee auf den ersten Blick wirkt, lassen sich damit doch für manche Probleme elegante Lösungen bauen. Welchen Preis das hat und welche Mittel von Java dabei hilfreich sind, ist der Gegenstand dieses Kapitels.
5. **Geschachtelte Klassen:** Klassendefinitionen können ineinander geschachtelt werden. Daraus ergeben sich interessante Möglichkeiten, darunter namenlose Klassen, von denen nur ein einziges Objekt existiert. Diese anonymen Klassen sind der Einstieg in die funktionale Programmierung, die derzeit rasch an Bedeutung gewinnt und in Java 8 Einzug hält.
6. **Nebenläufigkeit:** Threads sind der Schlüssel zu Java-Programmen, die mehrere Aufgaben gleichzeitig erledigen. Das eröffnet neue Möglichkeiten, bringt aber auch heikle Probleme mit sich. Dieses Kapitel stellt die fest in Java verankerten Sprachmittel vor, mit denen Nebenläufigkeit umgesetzt wird.
7. **Netzwerkprogrammierung:** Nach einer kurzen allgemeinen Einführung in Netzwerke zeigt dieses Kapitel, wie Java-Programme mit anderen Systemen über ein Netzwerk kommunizieren können. Das weist den Weg zu Java-Programmen, die ihre Dienste als Webserver zur Verfügung stellen und so mit beliebigen Browsern erreichbar sind.
8. **Reflection:** Java bietet die Möglichkeit, zur Laufzeit neue Klassen nachzuladen und deren Struktur programmatisch zu analysieren. In diesem Kapitel werden die Mittel und Wege vorgestellt, um mit dieser Technik sehr flexible Programme zu schreiben.
9. **Annotationen:** Annotationen sind formale Kommentare im Quelltext, die später wieder aufgespürt und ausgewertet werden können. Insbesondere kann der Java-Compiler mit neuen Klassen erweitert werden, die auf Annotationen reagieren und daraus beispielsweise schon während der Übersetzung neuen Quelltext generieren.

Die Auswahl der Themen ist zwangsläufig etwas subjektiv und ergibt sich aus meinen eigenen Erfahrungen.

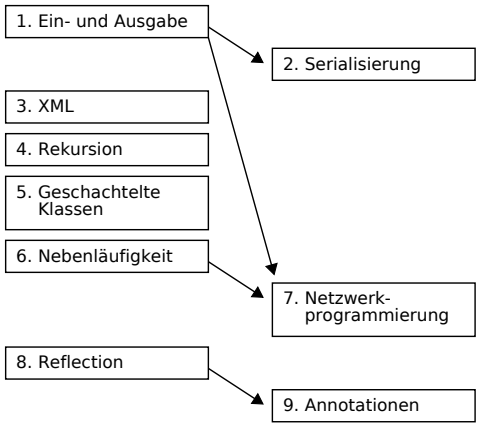
- Einige Kapitel befassen sich mit je einem konkreten Problemfeld und stellen die Art und Weise vor, wie Java damit umgeht.
- Andere Kapitel behandeln die Java-spezifische Umsetzung von eher konzeptionellen Themen, die zu interessanten Lösungswegen führen.
- Die restlichen Kapitel behandeln Java-spezifische Ansätze, die in dieser Form nicht in anderen Programmiersprachen zu finden sind.

Die folgende Skizze zeigt die Einordnung der Kapitel:



Kapitel-abhängigkeiten

Dieses Buch ist kein fortlaufender Text, der von vorne bis hinten gelesen werden will. Die Kapitel sind weitgehend isoliert und können einzeln und in fast beliebiger Reihenfolge durchgearbeitet werden. Ein paar Abhängigkeiten sind allerdings unvermeidlich, wie die folgende Skizze zeigt:



Die meisten Kapitel sind allerdings *in sich* zusammenhängend. Das schlägt sich in den Beispielprogrammen nieder, die meistens nach und nach ausgebaut werden.

Java-Versionen

Dieses Buch bezieht sich auf Java Version 7. Die Codebeispiele können auch auf Java-Versionen 6 und 5 zum Laufen gebracht werden, allerdings sind dazu an der einen oder anderen Stelle Anpassungen nötig.

Einige Abschnitte dieses Buches befassen sich mit geplanten Neuerungen von Java-Version 8, die voraussichtlich in der zweiten Hälfte des Jahres 2013 in einer endgültigen Fassung erscheinen wird. Trotzdem lässt sich ein Teil der neuen Sprachmittel

schon vorher erproben. Anhang D gibt Hinweise zu den erforderlichen Maßnahmen.

Software

Java-Entwicklungssystem

Um selbst Java-Programme zu entwickeln, brauchen Sie ein Java-Entwicklungssystem (*Java Development Kit*, JDK). Implementierungen des JDK 7 werden von verschiedenen Herstellern für verbreitete Betriebssysteme kostenlos zur Verfügung gestellt. Zur Wahl stehen beispielsweise die Implementierung der Oracle Corporation und der IBM Corporation. Ein Java-Laufzeitsystem (*Java Runtime Environment*, JRE) reicht dagegen *nicht* aus, weil es nur bereits fertige Java-Programme ausführen kann.

Das JDK wird in verschiedenen Ausgaben angeboten, die sich im Anwendungszweck unterscheiden. Passend für dieses Buch ist die Standard-Edition (Java SE), die zum Einsatz auf Desktop-Systemen gedacht ist. Im Gegensatz dazu ist die Enterprise-Edition (Java EE) für große Serversysteme gebaut.[2] Die *Programmiersprache* Java ist in allen Ausgaben dieselbe, lediglich die mitgelieferten Werkzeuge und Bibliotheken unterscheiden sich.

IDEs

Abgesehen vom Java-Entwicklungssystem ist der Einsatz einer integrierten Entwicklungsumgebung (*Integrated Development Environment*, IDE) sinnvoll. Eine ganze Auswahl davon ist frei verfügbar für nichtkommerzielle Zwecke, wie zum Beispiel Eclipse, Netbeans (Oracle), IntelliJ IDEA (JetBrains) und JDeveloper (Oracle). Der Funktionsumfang dieser IDEs reicht für die Zwecke dieses Buches vollkommen aus. Die konkrete Wahl ist weitgehend eine Sache der persönlichen Vorliebe.

Beispielprogramme

Dieses Buch enthält viele Beispielprogramme. Sie sind alle vollständig und lauffähig, auch wenn triviale und wiederkehrende Passagen manchmal aus Platzgründen nicht abgedruckt sind. Auf der Webseite zum Buch stehen alle ungekürzten Programme zum Download zur Verfügung. Der Code der Beispielprogramme ist auf den minimalen Umfang reduziert und möglichst von allem Ballast befreit, sodass der jeweils betrachtete Punkt klar erkennbar hervortritt.

Die abgedruckten Listings sind weitgehend unkommentiert. Sie stehen jeweils im Zusammenhang des Buchtextes, der die nötigen Erklärungen beisteuert.

---

[2] Die ebenfalls verfügbare Mikro-Edition (Java ME) für kleine Systeme (*embedded system*) verliert an Bedeutung. Zum einen sind selbst „kleine Systeme“ inzwischen leistungsfähig genug für die Standard-Edition. Zum anderen gewinnen konkurrierende Angebote, wie Googles Android, an Gewicht.

Hinweise für Dozenten

Der größte Teil dieses Buches dient als Leitlinie für die Vorlesung „Softwareentwicklung, Teil 2“ an der Fakultät 07 für Informatik und Mathematik der Hochschule München. Diese Lehrveranstaltung baut auf der Vorlesung „Softwareentwicklung, Teil 1“ auf, die die Voraussetzungen für diesen Stoff schafft. Einen Abriss der dort behandelten Themen finden Sie in Anhang A.

Zeitaufwand in einer Vorlesung

Die Vorlesung „Softwareentwicklung, Teil 2“ umfasst zwei Vorlesungen (je 90 Minuten) und ein Laborpraktikum (90 Minuten) pro Woche. In der Summe ergeben sich damit bei etwa dreizehn Semesterwochen 26 Einzelvorlesungen. Nach meiner Erfahrung nehmen die verschiedenen Kapitel etwa die folgende Zeit in Anspruch:

Kapitel	Thema	Anzahl Einzelvorlesungen
1	Ein- und Ausgabe	4
2	Serialisierung	2
3	XML	2
4	Rekursion	2
5	Geschachtelte Klassen	3
6	Nebenläufigkeit	3
7	Netzwerkprogrammierung	3
8	Reflection	2
9	Annotationen	3

Die einzelnen Kapitel bauen nicht alle aufeinander auf und können durchaus einzeln oder in Gruppen als Themenblöcke in eine Vorlesung eingebaut, anders arrangiert und mit anderen Themen gemischt werden. Sinnvollerweise sollten aber die auf Seite 7 dargestellten Abhängigkeiten berücksichtigt werden.

Es hat sich bewährt, die Beispielprogramme dieses Buches in der Vorlesung „vor laufender Kamera“ zu entwickeln. Der Stoff lässt sich damit plastischer und glaubhafter vermitteln als durch einen bloßen Vortrag. Zum Teil sind Programme so kurz und überschaubar, dass sie ohne großen Zeitverlust von Grund auf entwickelt werden können. Bei etwas längeren Programmen kann ein vorbereiteter Rahmen als Startpunkt dienen. Abgesehen davon schätzen es die Studierenden, wenn die Programme zur Nachbereitung als Download zur Verfügung gestellt werden.

Hinweise für Studierende

Die wahrscheinlich mit Abstand wichtigste Empfehlung für Studierende ist, die Programme in diesem Buch selbst nachzuvollziehen und die Ergebnisse zu verifizieren. Es reicht nicht aus, die abgedruckten Quelltexte durchzulesen und sich selbst zu versichern, dass sie wohl die zugesagten Ergebnisse liefern würden, wenn man sie tatsächlich ausführen würde. Natürlich ist auch mechanisches Abtippen oder Kopieren der heruntergeladenen Listings nicht sinnvoll.

Am besten arbeiten Sie jeweils einige Seiten durch, klappen dann das Buch zu und versuchen die betreffenden Programme selbst mit dem Wissen zu rekonstruieren, das Sie aufgenommen haben. Dabei hilft Ihnen eine Liste der Kurzbeschreibungen aller Programme, die Sie auf der Webseite zum Buch finden.

Die Codebeispiele im Text sind auf das Minimum gekürzt und umfassen nur die notwendige Funktionalität, um den jeweils betrachteten Gegenstand hinreichend zu illustrieren. Sie eignen sich in den meisten Fällen gut als Ausgangspunkt für Variationen und Erweiterungen. Experimentieren Sie mit den Programmen!

Webseite

Material zum Buch finden Sie auf der Webseite

```
http://sol.cs.hm.edu/4129
```

Dort steht der Text selbst in verschiedenen elektronischen Formaten zum Download zur Verfügung. Außerdem finden Sie auf dieser Seite eine Archivdatei mit allen Beispielprogrammen aus dem Buchtext.

Lösungen zu den Aufgaben im Buch sende ich auf Anfrage gerne an Lehrende, mache sie aber nicht allgemein verfügbar.

Lizenz

Die elektronische Fassung dieses Buchs sowie alle Beispielprogramme stehen unter der

> Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

Einzelheiten zu dieser Lizenz finden Sie auf

`http://creativecommons.org/licenses/by-nc-sa/3.0`

Kurz gefasst bedeutet diese Lizenz, dass dieses Material

- kopiert, verteilt und übertragen werden darf und
- nach Bedarf angepasst werden kann.

Die Bedingungen dafür sind, dass

- der Autor (Reinhard Schiedermeier) genannt,
- das Material weder ganz noch teilweise kommerziell genutzt und
- jedes Werk, das auf diesem Material aufbaut, unter der gleichen Lizenz veröffentlicht wird.

Dank

Diesen Text selbst haben meine sehr geschätzte Kollegin Ulrike Hammerschall und mein langjähriger Weggefährte Klaus Köhler mit gewohnt scharfem Blick durchgearbeitet. Auch meine Frau Gudrun Schiedermeier hat viel Aufwand investiert und nicht mit konstruktiver Kritik gespart. Schließlich hat die Lektorin Katharina Pieper zahllose Unklarheiten und Formulierungsschwächen aufgedeckt. Ihnen allen danke ich dafür, dass sie zu diesem Buch beigetragen haben.

Reinhard Schiedermeier
München, 2011/2012

Inhaltsverzeichnis

Kapitel

1 Ein- und Ausgabe (I/O)

Lernziele

In diesem Kapitel lernen Sie

- wie ein Java-Programm Eingaben von der **Tastatur lesen** und auf den **Bildschirm schreiben** kann und wie man die Eingaben aus Dateien holen und die Ausgaben in Dateien auffangen kann.
- auf welchen grundlegenden Methoden die **gesamte Ein- und Ausgabe** von Java aufgebaut ist.
- wie Sie **Dateien** lesen und schreiben können und wie sich durch Pufferung die Übertragungsgeschwindigkeit drastisch steigern lässt.
- wie Sie mit **Filterklassen** die Daten beim Lesen und Schreiben fast beliebig modifizieren können, beispielsweise um **Zip-Dateien** zu lesen und zu erzeugen und vieles mehr.
- dass das raffinierte **Entwurfsmuster** „Decorator“ hinter den I/O-Klassen steht und wie das Decorator-Pattern funktioniert.
- wie Java mit **Textdateien** umgeht und wie Text- und Binärdateien zueinander stehen.
- wie Sie selbst **neue I/O-Klassen** entwickeln und nahtlos mit den Bibliotheksklassen integrieren können.
- wie ein Java-Programm **Filesystem-Operationen** (verschieben, kopieren, löschen von Dateien und so weiter) steuern und wie es mit **Directories** arbeiten kann.

Vielfältige Aufgaben

Jedes Programm muss mit seiner Umgebung kommunizieren, sonst wäre es sinnlos. Dieses Kapitel stellt die verschiedenen Mechanismen vor, mit denen ein Java-Programm mit der Umgebung Kontakt aufnehmen kann. Im einfachsten Fall wird nur etwas Text ausgegeben, wie vom minimalen Erstprogramm „Hello World“. Im Allgemeinen können die Kommunikationspartner eines Programms aber sehr vielgestaltig sein, wie zum Beispiel

- ein menschlicher Benutzer, der im Dialog arbeitet,
- das Filesystem des Computers für längerfristig genutzte Daten,
- andere, gleichzeitig laufende Programme auf demselben Rechner oder
- Programme auf anderen Computern, die über ein Netzwerk verbunden sind.

Diese Vielfalt ist ein Grund dafür, dass die entsprechenden Mechanismen in Java auf den ersten Blick nicht ganz einfach zu überschauen und zu beherrschen sind.[1] Trotz der Komplexität erweist sich die Umsetzung in Java als recht leistungsfähig und flexibel.

Die Bezeichnungen „Eingabe“ und „Ausgabe“ beziehen sich auf den Standpunkt des Programms. *Eingabe* holt Daten von außen *in* das Programm, *Ausgabe* transportiert Informationen *aus* dem Programm in die Umgebung.

I/O über Bibliotheksklassen

Oft bezeichnet man die Gesamtheit der Sprachmittel zur Ein- und Ausgabe bloß mit dem Kürzel „I/O“ für *Input/Output*. In Java sind Interfaces und Klassen der Laufzeitbibliothek mit ihren jeweiligen Methoden für die Ein- und Ausgabe zuständig. Die meisten dieser Typen finden sich im Package `java.io`. Dementsprechend beschäftigt sich dieses Kapitel weniger mit der *Sprache* Java, sondern in erster Linie mit Bibliotheksklassen sowie ihren Beziehungen, Methoden und Einsatzmöglichkeiten. Eine Ausnahme ist die „automatische Ressourcenverwaltung“ (ARM, Seite 36), die mit Java 7 eingeführt wurde und die ein unvermeidliches, aber notorisch heikles Problem auf elegante und allgemeine Art löst.

1.1 Standardein- und -ausgabe

I/O mit minimalen Mitteln

Java bietet, ebenso wie andere Programmiersprachen, einen einfachen Weg, um mit minimalem Aufwand Eingabedaten in ein Programm zu holen und Ergebnisse wieder auszugeben. Diese **Standardein- und -ausgabe** reicht für einfache Anwendungen aus.

Mit Unterstützung des Betriebssystems kann die Standardein- und -ausgabe für eine interessante Art der Kopplung unabhängiger Programme („Filter“) genutzt werden. Aus Einzelprogrammen mit begrenzter Funktionalität können damit Lösungen für komplexe Probleme zusammengefügt werden.

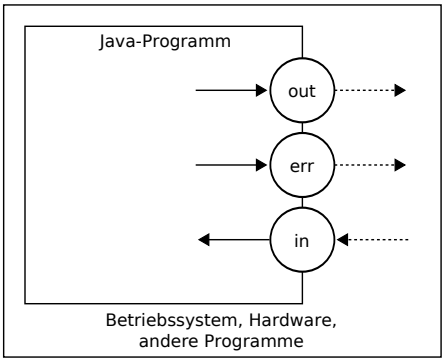
1.1.1 Vordefinierte I/O-Objekte

Klassenvariablen `System.in`, `.out`, `.err`

Die Standardein- und -ausgabe verläuft über drei vordefinierte, öffentliche Varia-

[1] Das ist kein spezifisches Problem von Java, sondern gilt für andere Programmiersprachen ebenso.

blen mit den Namen `in`, `out` und `err` in der Klasse `java.lang.System`. Diese drei Variablen repräsentieren gewissermaßen „Tore“ an der Grenze zwischen Java-Code und der Außenwelt. Was jenseits dieser Genze abläuft, ist Sache des Betriebssystems und unterliegt nicht der Kontrolle des Programms.



Technisch sind die drei Objekte `in`, `out` und `err` unveränderliche Klassenvariablen von `System`[2] . Die Definition von `System` kann man sich etwa folgendermaßen vorstellen:

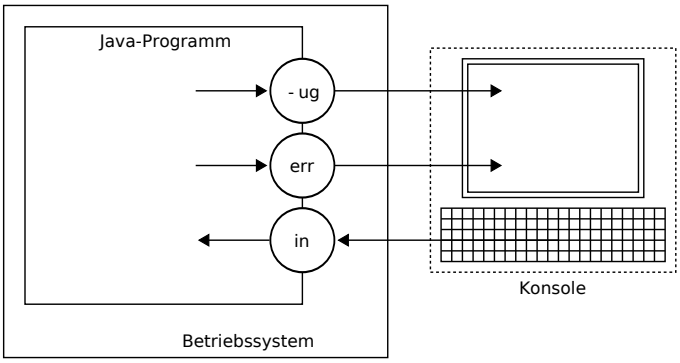
```
public final class System {
    public static final InputStream in = ...;
    public static final PrintStream out = ...;
    public static final PrintStream err = ...;
    ...
```

Tastatureingabe und Bildschirmausgabe

Die Typen der Variablen (`InputStream` und `PrintStream`) spielen im Moment noch keine Rolle. Wenn weiter keine Maßnahmen ergriffen werden, dann sind `in`, `out` und `err` mit Tastatur (`in`) und Bildschirm (`out` und `err`) verbunden. Die Kombination von Tastatur und Bildschirm bezeichnet man als „Konsole“.[3] Der Unterschied zwischen `out` und `err` wird weiter unten (Seite 26) erklärt.

[2] `System` ist eine Art Sammeltopf für allerlei Methoden mit systemnahen Aufgaben. Die Klasse enthält nur statische Variablen und Methoden. Sie kann weder instanziiert noch abgeleitet werden.

[3] Gelegentlich findet man daher auch den Begriff „Konsolen-Ein- und -Ausgabe“ als Synonym für Standardein- und -ausgabe.



Zeichenweises Lesen und Schreiben

Mit den beiden folgenden Methoden können einzelne Zeichen von `in` gelesen, beziehungsweise auf `out` und `err` geschrieben werden.

`int read() // System.in`
: Liefert als Ergebnis den Code des auf der Tastatur eingegebenen Zeichens im Bereich von 0 bis 255.

`void write(int code) // System.out, System.err`
: Gibt das Zeichen mit Code `code` auf dem Bildschirm aus.[4]

Das folgende Programm liest Tastatureingaben und kopiert sie auf den Bildschirm:[5]

```
import java.io.*;
import static java.lang.System.*;

public class EndlessConsoleEcho {
    public static void main(String... args) throws IOException {
        while(true) {
            int code = in.read();
            out.write(code);
        }
    }
}
```

Listing 1.1: Zeichenweises Kopieren der Standardeingabe auf die Standardausgabe bis zum Programmabbruch von außen.

[4] Zur Textausgabe kennen Sie sicher längst die Methoden `println`, `print` und `printf`. Diese sind weit komplexer, aber selbst auf der Grundlage von `write` implementiert.

[5] Die statische Import-Klausel erspart das Wiederholen von `System.` bei jedem Zugriff auf `System.in`, `System.out` und `System.err`. Die andere Import-Klausel ist nötig, um den Exceptiontyp `IOException` bekannt zu machen. Wildcard-Imports (Import-Klauseln mit „*“ am Ende) sind nicht unbedingt guter Programmierstil, halten aber die Codebeispiele kurz.

Programmabbruch mit Control-C

Das Programm läuft in einer Endlosschleife und endet nicht freiwillig. Allerdings stoppt die Tastenkombination Control-C[6] das Programm.[7] Diese Tastenkombination kommt nicht als Eingabe beim Java-Programm an. Das Betriebssystem erkennt Control-C, fängt diese Eingabe ab und bricht daraufhin das Programm ab.[8] Hier ein Ablaufbeispiel:

```
$ java EndlessConsoleEcho
hello?
hello?
anybody listening?
anybody listening?
^C
$
```

In diesem und den folgenden Beispielen wird als Prompt des Terminals stellvertretend das `$`-Zeichen verwendet. Dieses Prompt ist systemabhängig und kann bei Ihnen anders aussehen. Es hat keinen Einfluss auf den Ablauf von Java-Programmen. Im Gegensatz zu Systemausgaben sind die Benutzereingaben unterstrichen abgedruckt, sofern Verwechslungsgefahr besteht. Die Schreibweise „^C“ steht für die Tastenkombination Control-C.[9]

Fluchtwert -1 signalisiert Ende der Eingabe

Das Programm `EndlessConsoleEcho` (Listing 1.1) versucht endlos Eingaben zu lesen und muss gewissermaßen gewaltsam durch ^C gestoppt werden. Für ein kontrolliertes Ende sollte das Programm selbst erkennen, wenn die Eingabe endet. Das signalisiert der Rückgabewert -1 von `read`, der als **Fluchtwert** von den regulären Codewerten im Bereich 0 bis 255 eindeutig zu unterscheiden ist. Auf Seite 30 wird genauer darauf eingegangen.

Die folgende Erweiterung `ConsoleEcho` von `EndlessConsoleEcho` (Listing 1.1) kopiert so lange Tastendrücke auf den Bildschirm, bis die Eingabe beendet wird:

```
import java.io.*;
import static java.lang.System.*;
```

6 „Control-C“ bedeutet, dass die Control-Taste gehalten und dazu die Taste „C“ getippt wird. Die Control-Taste ist auf deutschen Tastaturen oft mit „Strg“ für „Steuerung“ beschriftet.

7 Diese Tastenkombination funktioniert in der Eingabeaufforderung von Windows und im Terminalfenster von Unix-Systemen, wie zum Beispiel Linux oder MacOS X.

8 Die Wahrheit ist komplizierter, wie immer. Unix schickt zum Beispiel ein Signal an das lesende Programm, das daraufhin endet. Das ist allerdings nur eine Voreinstellung, die geändert werden kann.

9 Auf vielen Systemen wird die Tastenkombination tatsächlich so dargestellt, um dem Benutzer eine lesbare Rückmeldung zu geben. Das ist allerdings nur eine Ersatzdarstellung des Systems zu diesem einen Zweck. Zum Beispiel kommt die Eingabe der beiden einzelnen Zeichen „^“ und „C“ als simpler Text im Programm an und hat nichts mit der Tastenkombination Control-C zu tun.

```
public class ConsoleEcho {
    public static void main(String... args) throws IOException {
        int code = in.read();
        while(code >= 0) {
            out.write(code);
            code = in.read();
        }
    }
}
```

Listing 1.2: Kopieren der Standardeingabe auf die Standardausgabe bis zum Eingabeende.

Ende der Tastatureingabe

Dabei stellt sich die Frage, wie die Eingabe via Tastatur überhaupt „beendet“ werden kann. Erneut kommt das Betriebssystem zu Hilfe: Es erkennt eine besondere Tastenkombination (Control-D auf Unix, Control-Z auf Windows)[10] und signalisiert dem lesenden Programm daraufhin das Ende der Eingabe. Im Java-Programm liefert `read` das Ergebnis -1, sobald der Unix-Benutzer Control-D eingibt:

```
$ java ConsoleEcho
hello?
hello?
anybody listening?
anybody listening?
^D
$
```

Obwohl dieses und das vorhergehende Ablaufbeispiel fast gleich aussehen, besteht dennoch ein wesentlicher Unterschied: `EndlessConsoleEcho` muss „von außen“ gestoppt werden, aber `ConsoleEcho` endet „freiwillig“. So könnte sich beispielsweise `ConsoleEcho` nach der Schleife noch mit einem Gruß verabschieden, während `EndlessConsoleEcho` dazu keine Gelegenheit mehr hat.

1.1.2 Redirection

Umlenkung der Standardein- und -ausgabe

Wie oben beschrieben, stehen die drei Objekte `System.in`, `out` und `err` an der Grenze eines Java-Programms und vermitteln Informationen von und nach außen. Dabei ist `System.in` mit der Tastatur verbunden, `System.out` und `err` mit dem Textbildschirm. Diese Kopplung spielt sich außerhalb des Java-Programms ab und ist aus dem Code heraus weder zu beeinflussen noch überhaupt erkennbar.

Die Zuordnung von `in`, `out` und `err` zu Tastatur und Bildschirm ist nur eine Voreinstellung und kann geändert werden. Diese Änderung spielt sich allerdings auf

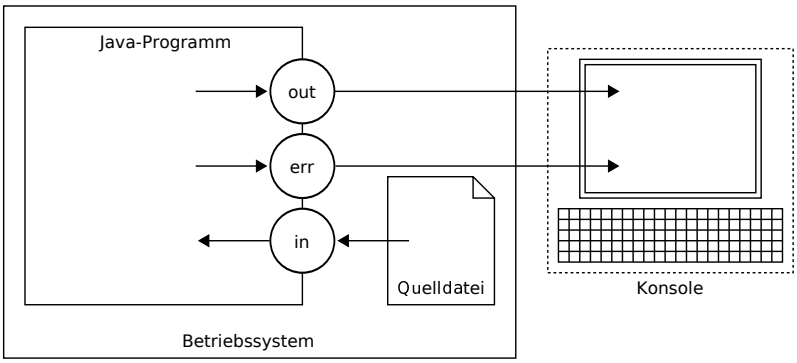
[10] Diese Tastenkombination wird vom Betriebssystem nur am Anfang einer Zeile abgefangen. Im Inneren einer Zeile gibt es keine Sonderbehandlung.

der Ebene des Betriebssystems ab und nicht im Java-Code. Dieser Abschnitt befasst sich folglich nicht mit Java, sondern mit Mitteln des Betriebssystems. Die hier gezeigten Mechanismen funktionieren sowohl in der Windows-Eingabeaufforderung wie auch im Unix-Terminal.

Eingabe-Umlenkung

Tastatureingabe aus einer Textdatei

Beim Start eines Java-Programms[11] kann festgelegt werden, woher `System.in` seine Daten bezieht. Wenn man nichts weiter unternimmt, wird `System.in` von der Tastatur gespeist, wie in den vorhergehenden Ablaufbeispielen gezeigt. Wenn man dagegen dem Programmaufruf auf der Kommandozeile ein < und einen Filenamen nachstellt, dann wird der Inhalt der betreffenden Datei über `System.in` an das Programm geliefert. `System.in` ist dann nicht mehr mit der Tastatur gekoppelt, sondern für die Dauer dieses Programmablaufs mit der angegebenen Quelldatei[12].



Eingabe-Umlenkung beim Programmaufruf

Die allgemeine Schreibweise lautet

```
programm < quelldatei
```

Im folgenden Beispiel wird `System.in` mit dem Quelltext des Programms selbst „gefüttert“:[13]

```
$ java ConsoleEcho < ConsoleEcho.java
import static java.lang.System.*;
```

[11] Das gilt für jedes Programm, nicht nur für Java-Programme.

[12] Mit „Quelldatei“ ist keine Quell*text*datei gemeint, sondern ein beliebiges File, dessen Inhalt als Quelle für die Standardeingabe dient. Die Quelldatei kann eine Binärdatei sein, auch wenn die Beispiele zur Veranschaulichung mit Text arbeiten.

[13] Die Ausgabe ist gekürzt wiedergegeben. Auf dem Bildschirm erscheint der gesamte Quelltext des Programms.

```
import java.io.*;
class ConsoleEcho {
...
}
$
```

Dieser Mechanismus wird als **Eingabe-Umlenkung** bezeichnet. Das Java-Programm liest dabei unverändert von `System.in` und hat keinen Hinweis darauf, dass hier kein Benutzer auf der Tastatur tippt, sondern dass die Eingaben aus einer Quelldatei stammen.[14]

Einsatzmöglichkeiten der Eingabe-Umlenkung

Diese Umlenkung bietet eine Reihe von Vorteilen:

- Ein Programm kann bei der Fehlersuche ohne Aufwand wiederholt mit denselben Eingaben gefüttert werden, ohne dass immer der gleiche Text eingetippt werden müsste.
- Bei Änderungen kann ein Programm mit vorbereiteten Eingaben getestet werden, die sich mit einem beliebigen Texteditor erstellen lassen. So kann sichergestellt werden, dass die Modifikationen keine unerwünschten Nebenwirkungen nach sich ziehen.
- Auf diese Art können weit mehr Daten an ein Programm geliefert werden, als es über eine Tastatureingabe praktikabel wäre.
- Eingaben können schnell umgesetzt werden. Letztlich begrenzen nur noch Betriebssystem und Programmcode den Datendurchsatz, nicht mehr die Fingerfertigkeit des Benutzers.

Programm zum Zeichen- und Zeilenzählen

Eine nützliche Variation von `ConsoleEcho` (Listing 1.2) ist das Programm `LineCharCounter`, das die Anzahl der Zeichen und Zeilen der Eingabe abzählt und am Ende ausgibt.[15]

```
import java.io.*;
import static java.lang.System.*;

public class LineCharCounter {
    public static void main(String... args) throws IOException {
        int lines = 0;  // Zeilenzähler
        int chars = 0;  // Zeichenzähler
```

[14] Das Programm könnte den Unterschied erkennen, wenn es die Geschwindigkeit der Eingabe analysieren würde. Bei Eingabe-Umlenkung treffen die Daten viel schneller ein, als ein Mensch sie tippen könnte.

[15] Das Programm vergleicht Codes mit `'\n'`, um Zeilenwechsel zu erkennen. Das funktioniert zwar auf Windows, Linux und MacOS X, ist aber dennoch nicht ganz sicher, weil nicht alle Systeme Zeilenwechsel auf diese Art codieren müssen.

```
        int code = in.read();
        while(code >= 0) {
            chars++;
            if(code == '\n')
                lines++;
            code = in.read();
        }

        out.printf("%d Zeichen, %d Zeile(n)%n", chars, lines);
    }
}
```

Listing 1.3: Abzählen und Protokollausgabe der Anzahl Zeichen und Zeilen der Eingabe.

Ein Test ohne Eingabe-Umlenkung zeigt das erwartete Verhalten. Beachten Sie, dass auch die Eingabetaste als Zeichen in das Programm gelangt und dort mitgezählt wird, nicht aber die Tastenkombination Control-D.

```
$ java LineCharCounter
hello?
^d
7 Zeichen, 1 Zeile(n)
```

Ein neuer Aufruf mit dem eigenen Quelltext als Quelldatei liefert dessen Umfang:

```
$ java LineCharCounter < LineCharCounter.java
786 Zeichen, 33 Zeile(n)
```

Beispiel: Tabulatoren durch Leerzeichen ersetzen

Nützlicher ist das folgende Programm `Detab`, das Tabulatorzeichen durch eine Anzahl Leerzeichen ersetzt. Die Spaltenbreite der Tabulatoren kann auf der Kommandozeile angegeben werden. Ohne Angabe wird eine voreingestellte Breite von 8 Stellen verwendet. Die selten verwendete innere `do`-Schleife stellt sicher, dass jedes Tabulatorzeichen zu wenigstens einem Leerzeichen expandiert wird.

```
import java.io.*;
import static java.lang.System.*;

public class Detab {
    public static void main(String... args) throws IOException {
        int tabWidth = args.length > 0? Integer.parseInt(args[0]): 8;
        int length = 0;     // aktuelle Zeilenlänge

        int code = in.read();
        while(code >= 0) {
            if(code == '\n') {
                out.write(code);
                length = 0;
            }
```

```
            else if(code == '\t')
                do {
                    out.write(' ');
                    length++;
                }
                while(length%tabWidth > 0);
            else {
                out.write(code);
                length++;
            }
            code = in.read();
        }
    }
}
```

Listing 1.4: Umwandlung von Tabulatorzeichen in Leerzeichen.

Mit diesem Programm können Tabulatoren aus Quelltext entfernt werden, wie zum Beispiel:[16]

```
$ java Detab < Detab.java
...
```

Hier wird der eigene Quelltext wieder ausgegeben. Möglicherweise ist auf den ersten Blick keine Änderung erkennbar. Die Wirkung des Programms wird aber deutlich, wenn die Tabulator-Spaltenbreite versuchsweise auf einen übertrieben hohen, wenn auch praktisch unbrauchbaren Wert gesetzt wird:

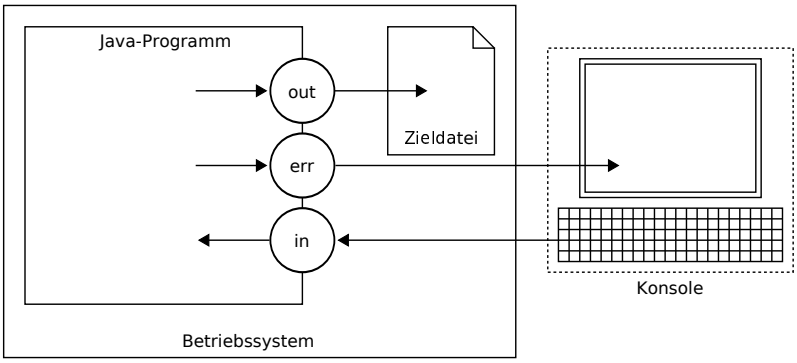
```
$ java Detab 32 < Detab.java
...
```

Ausgabe-Umlenkung

Bildschirmausgabe in eine Datei

Ebenso wie die Standardeingabe lässt sich auch die Standardausgabe umlenken (*output redirection*). Statt auf den Bildschirm werden die Ausgaben in eine Zieldatei geschrieben.

[16] Dieses Beispiel setzt voraus, dass in `Detab.java` tatsächlich Tabulatorzeichen vorkommen. Die meisten Editoren fügen automatisch Tabulatoren ein.



Die allgemeine Schreibweise lautet

```
programm > zieldatei
```

Die Zieldatei wird neu erzeugt. Doch Vorsicht! Sollte die Zieldatei schon vor dem Programmaufruf existieren, so wird sie kommentarlos ersetzt. Der Inhalt der alten Datei geht dabei verloren. Das folgende Beispiel kopiert die Tastatureingabe in die Zieldatei `output.txt`:

```
$ java ConsoleEcho > output.txt
Hi there!
^d
$
```

In `output.txt` steht jetzt eine Zeile mit dem Inhalt „Hi there!“. Das kann mit einem Texteditor überprüft werden oder mit einem Aufruf von `ConsoleEcho` (Listing 1.2):

```
$ java ConsoleEcho < output.txt
Hi there!
```

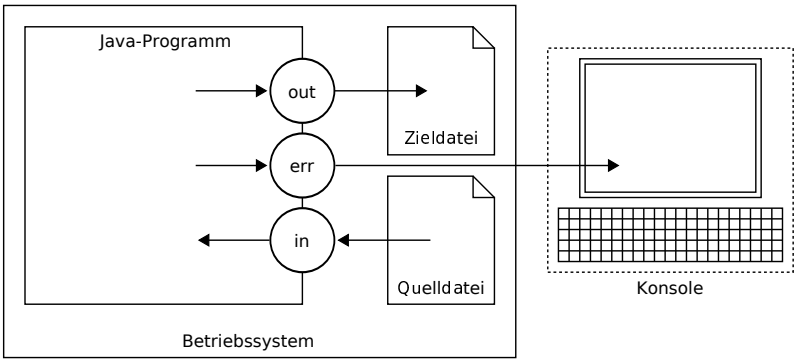
Ein- und Ausgabe-Umlenkung

Standard-I/O ohne Tastatur und Bildschirm

Die Umlenkungen der Standardein- und -ausgabe lassen sich kombinieren. Unverändert arbeitet das Java-Programm mit `System.in` und `out`. Tatsächlich wird allerdings weder die Tastatur noch der Bildschirm benutzt. Stattdessen stammen die Eingaben aus einer Quelldatei und die Ausgaben werden in eine Zieldatei geschrieben:[17]

```
programm < quelldatei > zieldatei
```

[17] Die Reihenfolge der Angaben der Umlenkungen spielt keine Rolle.



Beispiel: Kopieren einer Datei

Damit kann zum Beispiel ein File in ein anderes Directory kopiert werden: [18]

```
$ java ConsoleEcho < ConsoleEcho.java > /tmp/ConsoleEcho.java
```

Im Directory `/tmp` findet sich jetzt eine Kopie der Datei `ConsoleEcho.java`.[19]

1.1.3 Standard-Fehlerausgabe

`System.err` für Fehlermeldungen

Die bisher gezeigten Beispiele verwendeten immer `out` zur Standardausgabe. Das alternative Objekt `err` funktioniert weitgehend genau so wie `out`, hat allerdings einen anderen Zweck. Wie der Name andeutet, ist `err` nicht für reguläre Ausgaben gedacht, sondern für Ausgaben im Fehlerfall oder anderen Situationen, in denen das Programm nicht planmäßig arbeiten kann. Ausgaben auf `err` werden als **Standard-Fehlerausgabe** bezeichnet.

Modifiziert man `ConsoleEcho` (Listing 1.2) durch Austausch von `out` durch `err` zu `ConsoleErrorEcho`, so zeigt sich zunächst keine Wirkung:

```
$ java ConsoleErrorEcho
hello?
hello?
^d
$
```

Die Änderung wird sichtbar, wenn man versucht, die Ausgabe umzulenken:

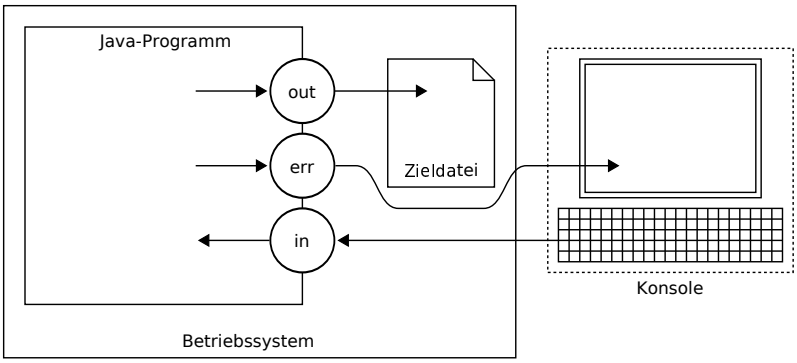
---

[18] Die Schreibweise von Pfadnamen ist betriebssystemabhängig. Auf einem Windows-Rechner wäre ein entsprechendes File `C:\windows\Temp\ConsoleEcho.java`.

[19] Diese Art Dateien zu kopieren ist ziemlich langsam. Das zeigt sich deutlich bei dem Versuch, eine Datei von einigen Megabyte Länge zu kopieren.

```
$ java ConsoleErrorEcho > output.txt
hello?
hello?
^d
$
```

Ausgaben auf `err` lassen sich nicht ohne Weiteres umlenken, sondern erscheinen immer auf dem Bildschirm.[20] Das entspricht dem beabsichtigten Einsatzzweck von `System.err`: Selbst bei aktiver Umlenkung hat das Programm die Möglichkeit, auf Probleme hinzuweisen. Reguläre Ausgaben und Fehlerausgaben nehmen unterschiedliche Wege und geraten nicht durcheinander.



Beispiel: Warnung bei leerer Eingabe

Die beiden Arten von Ausgaben lassen sich im gleichen Programm kombinieren. Das folgende Programm `ConsoleNonemptyEcho` warnt den Benutzer auf der Standard-Fehlerausgabe vor einer leeren Eingabe. Abgesehen davon arbeitet es genauso wie `ConsoleEcho` (Listing 1.2). Das Flag `empty` signalisiert, ob jemals etwas ausgegeben wurde.

```
import java.io.*;
import static java.lang.System.*;

public class ConsoleNonemptyEcho {
    public static void main(String... args) throws IOException {
        boolean empty = true;
        for(int code = in.read(); code >= 0; code = in.read()) {
            out.write(code);
            empty = false;
        }
        if(empty)
            err.println("Empty input!");
    }
}
```

Listing 1.5: Ausgabe auf die Standard-Fehlerausgabe bei leerer Eingabe.

[20] Mit dem Operator `2>` lässt sich auch die Standard-Fehlerausgaben umlenken.

Wenn man das Programm startet und mit etwas Texteingaben versorgt, kopiert es diese ohne Warnung in die Zieldatei:

```
$ java ConsoleNonemptyEcho > output.txt
Hello there!
^d
$
```

Beendet man die Eingabe sofort nach dem Programmstart, so erscheint trotz Ausgabe-Umlenkung eine Warnung auf dem Bildschirm. Unabhängig davon wird eine Zieldatei erzeugt, die aber wie erwartet leer bleibt.

```
$ java ConsoleNonemptyEcho > output.txt
^d
Empty input!
$
```

1.2 Byteströme

Byteströme unbekannter Länge

Die Ein- und Ausgabe allgemeiner Daten wird in Java über **Byteströme** abgewickelt. Byteströme sind lineare, flache Folgen von Bytes, die alle gleich behandelt werden. Es gibt keine inneren Strukturen in einem Bytestrom, wie etwa Abzweigungen oder abgegrenzte Teilfolgen. Die Länge eines Bytestroms ist in der Regel nicht bekannt. Man kann insbesondere nicht davon ausgehen, dass er komplett in den Speicher passt und beispielsweise zur Verarbeitung in ein Array kopiert werden kann. Schließlich können Byteströme im Allgemeinen nur sequenziell verarbeitet werden, das heißt vom Anfang bis zum Ende.

1.2.1 Abstrakte Basisklassen

Abstrakte Basisklassen für Byteströme

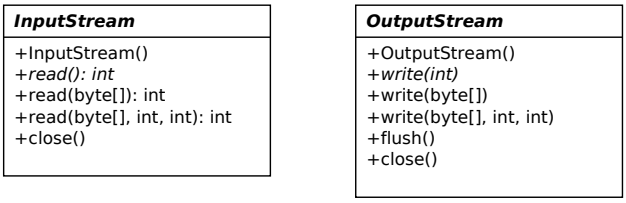
Die beiden abstrakten Basisklassen (*Abstract Base Classes*, ABCs) `InputStream` und `OutputStream` sind die Grundlage für alle Arten von Byteströmen, die von außen in das Programm beziehungsweise vom Programm nach außen führen:

`InputStream`
: Eingabe-Bytestrom, liefert Bytes aus einer Quelle.

`OutputStream`
: Ausgabe-Bytestrom, schreibt Bytes in eine Senke.

Beides sind ABCs, das heißt, es gibt keine Objekte dieser Klassen selbst. `System.in` und `System.out` sind Beispiele von Objekten konkreter Klassen, die von den ABCs abgeleitet sind.

`InputStream` und `OutputStream` sind im Package `java.io` definiert, ebenso wie fast alle weiteren Klassen und Interfaces in diesem Kapitel. `InputStream` und `OutputStream` haben keine gemeinsame Basisklasse.[21]



1.2.2 Lesen und Schreiben

Lesen mit `read`, Schreiben mit `write`

Die beiden wichtigsten Methoden von Byteströmen wurden schon im Zusammenhang mit der Standardein- und -ausgabe (Seite 17) vorgestellt:

`int read() // InputStream`
: Liefert das nächste Byte des Eingabe-Bytestroms oder -1, wenn der Eingabe-Bytestrom beendet ist.

`void write(int code) // OutputStream`
: Schreibt das Byte `code` auf den Ausgabe-Bytestrom.

Kopien von Byteströmen

Das Programm `ConsoleEcho` (Listing 1.2) lässt sich zum Programm `StreamCopy` verallgemeinern, das zunächst immer noch die Standardein- auf die Standardausgabe kopiert.

```
import java.io.*;

public class StreamCopy {
    public static void main(String... args) throws IOException {
        InputStream input = System.in;
        OutputStream output = System.out;
        int code = input.read();
        while(code >= 0) {
            output.write(code);
```

[21] Abgesehen von der trivialen Basisklasse `Object`.

```
            code = input.read();
        }
    }
}
```

Listing 1.6: Zeichenweises Kopieren eines `InputStream` auf einen `OutputStream`.

In `StreamCopy` werden die Standard-I/O-Objekte, `System.in` und `System.out`, nicht mehr direkt angesprochen, sondern an zwei lokale Variablen `input` und `output` der Typen `InputStream` und `OutputStream` zugewiesen. Das ist die einzige Änderung gegenüber `ConsoleEcho` (Listing 1.2).

Ergebnistyp `int` von `read`

`read` liefert als reguläres Ergebnis einen Wert zwischen 0 und 255. Der Ergebnistyp ist `int`, obwohl man vielleicht `byte` erwartet hätte. Der Grund für `int` ist das Ende der Eingabe, das `read` mit dem Fluchtwert -1 signalisiert. Dieser Fluchtwert ist eindeutig als solcher erkennbar, weil -1 keinesfalls im Bytestrom vorkommen kann. Wenn der Ergebnistyp von `read` nur `byte` wäre, dann könnte man keinen Fluchtwert festlegen, weil jeder `byte`-Wert im Eingabe-Bytestrom vorkommen kann! Zur Rückgabe wird also ein „größerer" Typ als `byte` gebraucht, der den ganzen Wertebereich von `byte` umfasst und der außerdem noch Raum für einen Fluchtwert lässt. Die Wahl fiel auf `int`, das Arbeitspferd der ganzzahligen Typen.

Keine Exception am Eingabeende

Eine Exception am Ende der Eingabe wäre keine akzeptable Alternative: Das gesamte Exception-Handling befasst sich mit unerwarteten Situationen, in die ein normal arbeitendes Programm nicht geraten sollte. Eine JVM muss sich aus diesem Grund nicht viel Mühe damit geben, Exceptions effizient abzuwickeln. Schließlich befindet sich das Programm ohnedies schon im Trudeln und hat andere Probleme, als besonders performant abzustürzen. Auf der anderen Seite ist das Ende einer Eingabe bestimmt keine völlig abwegige Situation, mit der nicht zu rechnen wäre. Schließlich enden ja fast alle Eingaben früher oder später!

`int`-Argument von `write`

Aus Symmetriegründen akzeptiert `write` ebenfalls ein `int`-Argument, gibt aber ein Byte mit dem entsprechenden Wert aus. Vom `int`-Argument werden nur die niederwertigen 8 Bits berücksichtigt und die 24 höherwertigen Bits ignoriert. Die folgenden Aufrufe führen daher alle zur gleichen Ausgabe, weil nur die höherwertigen 24 Bits der Argumente abweichen und die niederwertigen 8 Bits jeweils gleich sind:[22]

```
write(255)                  // 255 = 0x000000FF
write(-1)                   // -1 = 0xFFFFFFFF
write(Integer.MAX_VALUE)    // Integer.MAX_VALUE = 0x7FFFFFFF
write(305420031)            // 305420031 = 0x123456FF
```

[22] Im Kommentar ist jeweils die hexadezimale Darstellung des Arguments abgedruckt.

Ein `write`-Aufruf mit einem `byte`-Argument erfordert keine Sonderbehandlung. Das Argument wird implizit in einen temporären `int`-Wert konvertiert und von diesem nur das niederwertige Byte ausgegeben, also genau das ursprüngliche Byte.

Blockierende Aufrufe

Langsame `read`- und `write`-Aufrufe blockieren

Die Geschwindigkeit, mit der ein Programm eingehende Daten verarbeitet, muss nicht mit der Geschwindigkeit übereinstimmen, mit der die Quelle Daten liefert. Wenn das lesende Programm `read`-Aufrufe schneller absetzt, als der Eingabe-Bytestrom Bytes zur Verfügung stellen kann, dann blockieren `read`-Aufrufe so lange, bis neue Daten verfügbar sind oder das Ende des Eingabe-Bytestroms signalisiert wird. Das kann viele Ursachen haben, wie zum Beispiel ein langsamer Datenträger, ein ausgelastetes System, eine schmalbandige Netzwerkverbindung oder ein Mensch, der Text auf der Tastatur eingibt.

Das Gleiche gilt für die Ausgabeseite: `write`-Aufrufe blockieren so lange, bis die Daten abgesetzt werden konnten.

▶ Größere Programme haben oft verschiedene Aufgaben zu erledigen. Methodenaufrufe, die auf unbestimmte Zeit blockieren, stören dabei, weil das Programm in dieser Zeit nichts anderes tun kann. Eine Lösung für einfache Fälle bietet die `InputStream`-Methode

`available` schätzt verfügbare Daten ab

`int available()`
: Liefert die Anzahl Bytes, die jetzt sofort *ohne Blockieren* gelesen werden könnten.

`available` liefert nicht unbedingt die maximale Anzahl verfügbarer Bytes, sondern nur eine defensive Abschätzung. In Wahrheit könnten viel mehr Daten vorliegen.

Ein Dilemma tritt auf, wenn `available` das Ergebnis null zurückgibt. Diese Auskunft kann zweierlei bedeuten:

1. `read` darf nicht aufgerufen werden, weil keine Bytes verfügbar sind und daher die Gefahr des Blockierens besteht.
2. Die Eingabe ist beendet und es können *nie wieder* Daten gelesen werden. Um das herauszufinden, müsste `read` aufgerufen und das Ergebnis -1 beobachtet werden.

Wegen Punkt 1 darf `read` nicht aufgerufen werden, wegen Punkt 2 muss es aufgerufen werden!

Insgesamt bietet `available` keinen allzu verlässlichen Weg, um Blockieren zu vermeiden. Bessere Lösungen bauen auf Threads, mit denen sich Kapitel 6 befasst. ◀

Exceptions

Fehler beim Lesen und Schreiben

`read` und `write` können mit einer `IOException` scheitern, für die es vielfältige Ursachen gibt. Eine `IOException` zeigt einen ziemlich unspezifischen Ein-/Ausgabefehler an. Die Klasse ist Basisklasse für eine Vielzahl abgeleiteter Exceptionklassen, die konkretere Ursachen ausdrücken. Aus der Sicht der ABCs können diese Ursachen aber nicht genauer eingegrenzt werden, daher werfen die Methoden von `InputStream` und `OutputStream` nur `IOExceptions`, aber keine spezielleren Exceptions.

1.2.3 Pufferung

Sammlung von Bytes in Paketen

Am früheren Beispielprogramm `EndlessConsoleEcho` (Listing 1.1) lässt sich beobachten, dass das Echo nicht sofort bei jedem Tastendruck erscheint, sondern erst nachdem die Eingabetaste gedrückt wird. Die Ursache für dieses Verhalten ist die Pufferung des Bytestroms. Der Transport von Daten aus dem Programm zum endgültigen Ziel erfordert einigen Aufwand seitens des Betriebssystems, ebenso wie in der Gegenrichtung der Transport von Daten von der eigentlichen Quelle bis in das Programm. Dieser Aufwand lohnt sich nicht für ein einzelnes oder nur wenige Bytes. Auf dem Weg zwischen Programm und Quelle beziehungsweise Ziel werden Daten deshalb in Päckchen gesammelt, die erst bei einem lohnenden Umfang tatsächlich auf die Reise geschickt werden. Dieses Verfahren wird als **Pufferung** bezeichnet.

Lesen auf Vorrat

In der Regel werden sowohl Eingaben wie auch Ausgaben gepuffert. Wenn ein Programm mit einem `read`-Aufruf ein einzelnes Byte anfordert, wird möglichst gleich ein ganzer Block von Daten von der Quelle geholt. Das erste Byte dieses Blocks wird sofort an den Aufrufer zurückgegeben, der Rest für künftige `read`-Aufruf in einem Eingabepuffer als Vorrat zur Seite gelegt. Beim nächsten `read`-Aufruf wird zuerst der verbleibende Vorrat überprüft. Falls vorhanden, erhält das Programm das nächste Byte direkt aus dem Vorrat. Erst wenn der Vorrat aufgebraucht ist, wird ein neuer Block gelesen. Die Anzahl der tatsächlichen Zugriffe auf die Datenquelle ist also weit niedriger als die der einzelnen `read`-Aufrufe.

Schreiben in Blöcken

In der Gegenrichtung wird nicht jedes Byte sofort zur Datensenke durchgestellt. Stattdessen werden Bytes in einem Ausgabepuffer aufgestaut. Erst wenn der Ausgabepuffer voll ist, wird er komplett zur Datensenke übermittelt und steht anschließend wieder zur Verfügung für weitere Ausgaben. Wie bei der Eingabe gibt es viel weniger Zugriffe auf die Datensenke, als das Programm `write`-Aufrufe ausführt.

Abgrenzung von Paketen

Die Arbeitsweise der Pufferung hängt von der einzelnen Datenquelle beziehungsweise -senke ab. Während allgemeine `OutputStreams` beispielsweise einfach eine gewisse Anzahl Bytes aufsammeln, reagiert `System.out` auf Zeilenwechsel. Das ist der Grund für das zeilenweise Echo in `EndlessConsoleEcho` (Listing 1.1).[23]

Kontrolle der Ausgabepufferung

Für ein Java-Programm ist die Pufferung nicht direkt erkennbar.[24] Die Pufferspeicher selbst sind verborgen. Ausgabeseitig kann ein Java-Programm „auf Verdacht" die Leerung des Ausgabepuffers mit einem Aufruf der Methode `flush` erzwingen:[25]

`void flush()`
: Entleert die Ausgabepuffer und gibt alle gepufferten Daten sofort aus.

Beispiel: langsame Textausgabe

Das folgende Programm `SlowLetters` gibt endlos die Buchstaben des Alphabets mit einer Rate von 10 Zeichen pro Sekunde aus:[26]

```
import java.io.*;

public class SlowLetters {
    public static void main(String... args) throws IOException, InterruptedException {
        int letter = 'A';
        while(true) {
            System.out.write(letter);
            // out.flush();
            Thread.sleep(100);
            letter++;
            if(letter > 'Z')
                letter = 'A';
        }
    }
}
```

Listing 1.7: Beobachtung der Pufferung der Standardausgabe.

Startet man das Programm, so erscheint zunächst nichts auf dem Bildschirm, weil sich die Buchstaben in einen Ausgabepuffer ansammeln. Nach einigen Sekunden

[23] Ein weiterer Grund ist die Eingabepufferung. Das Betriebssystem schickt nicht jeden Tastendruck sofort zum Programm, sondern sammelt sie erst an, bis die Eingabetaste gedrückt wird.

[24] Eine indirekte Testmöglichkeit wäre eine genaue Messung des Zeitverhaltens einzelner Methodenaufrufe.

[25] Eingabeseitig gibt es keine vergleichbare Möglichkeit.

[26] Der Aufruf `Thread.sleep(100)` blockiert für 100 Millisekunden. In dieser Zeit tut das Programm nichts, außer auf den Ablauf der Zeitspanne zu warten. Es verbraucht dabei auch keine Rechenleistung. `sleep` kann eine `InterruptedException` werfen, die hier nicht behandelt wird. Die Methode wird auf Seite 384 genauer besprochen.

ist der Puffer gefüllt und es wird ein kompletter Block[27] von Buchstaben auf einmal ausgegeben. Dieses schubweise Schreiben wiederholt sich immer dann, wenn der Ausgabepuffer voll ist. Wenn man den `flush`-Aufruf aktiviert, dann werden die einzelnen Buchstaben sofort und in gleichmäßiger Geschwindigkeit ausgegeben.

An dieser Stelle gibt es noch keinen zwingenden Grund für den Einsatz von `flush`. Abgesehen von den zeitlichen Verhältnissen läuft ein Programm bisher mit und ohne `flush` gleich ab. Das Bild ändert sich in Kapitel 7 (Seite 447). Dort sehen wir Code, in denen der Aufruf von `flush` entscheidende Bedeutung gewinnt und über Funktionieren oder Scheitern der Programme bestimmt.

1.2.4 Schließen von Streams

Betriebssystem-Ressourcen für Streams

Der Zweck der Ein- und Ausgabe ist der Datenaustausch mit Quellen und Senken, die außerhalb des Java-Programms liegen.[28] Für die Anlieferung und den Abtransport der Daten ist das zugrunde liegende Betriebssystem zuständig, das dazu Verwaltungsdaten braucht. Solche Verwaltungsdaten fallen, ebenso wie Puffer, unter den Begriff **Ressourcen**[29] .

Freigabe von Betriebssystem-Ressourcen

Ein Bytestrom bindet also Ressourcen des Betriebssystems, das heißt, er verbraucht Platz auch außerhalb des Java-Programms. Das Betriebssystem kann allerdings nicht wissen, wie lange ein Bytestrom tatsächlich benutzt wird. Deshalb muss sich ein Programm selbst dazu äußern.

Notwendigkeit von `close`-Aufrufen

Die beiden Klassen `InputStream` und `OutputStream` definieren je eine Methode `close`, die nach Gebrauch der Objekte aufgerufen werden muss:

`void close()`
: Gibt die gebundenen Ressourcen frei.

Bei Ausgabe-Byteströmen sorgt `close` vor dem Abbau der Verbindung außerdem noch dafür, dass eventuell gepufferte Daten an ihr Ziel transportiert und dort sicher abgeladen werden. Technisch ruft `close` automatisch `flush` auf.

`close` als letzte Methode

Sobald `close` aufgerufen wurde, ist der Bytestrom geschlossen. Jeder nachfolgende

[27] Das genaue Verhalten ist systemabhängig. Die Größe der Blöcke ist nicht auf allen Systemen gleich.

[28] Auch Byte-Arrays können in Streams verpackt und dann mit I/O-Mechanismen verarbeitet werden. In diesem Fall ist keine externe Datenquelle oder -senke im Spiel. Allerdings ist das eher ein Sonderfall.

[29] Ressourcen sind alle Arten von begrenzten Betriebsmitteln, wie zum Beispiel auch Rechenleistung, Netzwerkbandbreite und Massenspeicherplatz. Hier geht es um Hauptspeicher für Datenstrukturen des Betriebssystems.

Versuch Daten zu lesen oder zu schreiben endet mit einer `IOException`. Ein einmal geschlossener Bytestrom kann nicht wieder geöffnet werden. Er ist endgültig verbraucht.

Im Programm `StreamCopy` (Listing 1.6) fehlen die `close`-Aufrufe noch. Das folgende Programm ergänzt sie:

```
import java.io.*;

public class StreamCopyClosing {
    public static void main(String... args) throws IOException {
        InputStream input = System.in;
        OutputStream output = System.out;

        int code = input.read();
        while(code >= 0) {
            output.write(code);
            code = input.read();
        }

        input.close();
        output.close(); // implizites flush
    }
}
```

Listing 1.8: Lesen und Schreiben allgemeiner Streams.

Allerdings stellt sich die Frage, wie `StreamCopy` (Listing 1.6) und die anderen Beispielprogramme im Abschnitt 1.1 eigentlich funktionieren konnten. Dort fehlen schließlich alle `close`-Aufrufe! Tatsächlich zählen die drei Standard-I/O-Objekte zu den wenigen Ausnahmen, bei denen `close`-Aufrufe nichts bewirken und ohne Schaden ausgelassen werden können, *falls* keine I/O-Redirection im Spiel ist. Weil aber im Java-Programm eine Umlenkung nicht erkennbar ist, sind `close`-Aufrufe auch bei den Standard-I/O-Objekten Pflicht.

Probleme fehlender `close`-Aufrufe

Das folgende einfache Beispiel demonstriert die Folgen eines fehlenden `close`. Wie auf Seite 26 wird `ConsoleEcho` (Listing 1.2) mit I/O-Redirection benutzt, um eine Datei zu kopieren. Anders als auf Seite 26 wird hier aber nicht die Quelltext-, sondern die *Bytecode-Datei* kopiert:

```
$ java ConsoleEcho < ConsoleEcho.class > /tmp/ConsoleEcho.class
```

Ein Vergleich von Original und Kopie zeigt, dass Letztere kürzer und damit unvollständig ist. Die Zahlen am Zeilenanfang weisen die Dateilänge in Bytes aus.

```
$ ls -l ConsoleEcho.class /tmp/ConsoleEcho.class
737 ConsoleEcho.class
731 /tmp/ConsoleEcho.class
```

Die Standardein- und -ausgabe puffert auf Zeilenbasis. Eine Quelltextdatei schließt in der Regel mit einem Zeilenwechsel ab.[30] Die letzte Zeile wird als voller Puffer am Ende komplett übertragen. Für die Bytecode-Datei gilt das nicht. Bei ihr geht der letzte Puffer verloren und fehlt in der Ausgabe. Der Fehler verschwindet, wenn man in `ConsoleEcho` (Listing 1.2) die `close`-Aufrufe einfügt.

Fehlende `close`-Aufrufe können zu tückischen Problemen führen, die sich nur schwer reproduzieren lassen. Typischerweise funktioniert ein Programm beim Entwickler fehlerfrei, stürzt aber beim Kunden nach einer Weile ab, besonders wenn es unter Last gerät.

1.2.5 ARM

`close`-Aufrufe in `finally`

Wie im vorhergehenden Abschnitt deutlich wurde, sollte ein Programm unbedingt alle Streams schließen, sobald sie nicht mehr gebraucht werden. In einem ersten Ansatz könnte man beispielsweise einen `finally`-Block anfügen, der ja zuverlässig ausgeführt wird:

```
import java.io.*;

public class StreamCopyFinally {
    public static void main(String... args) throws IOException {
        InputStream input = System.in;
        OutputStream output = System.out;
        try {
            for(int code = input.read(); code >= 0; code = input.read())
                code = input.read();
        }
        finally {
            input.close();
            output.close();
        }
    }
}
```

Listing 1.9: Kopieren von Streams mit `close`-Aufruf in einem `finally`-Block.

Diese Lösung ist allerdings unvollständig: Zwar werden die `close`-Aufrufe nach dem `try`-Block mit Sicherheit erreicht, aber sie können selbst mit einer `IOException` scheitern. Folglich müssen sie in einen weiteren `try/catch`-Block innerhalb von `finally` gepackt werden. Aber auch dann kann es passieren, dass beim Scheitern des ersten `close`-Aufrufs der zweite ausgelassen wird. Die ganze Konstruktion

[30] Dafür sorgen im Allgemeinen schon Texteditoren. Mit etwas Mühe lässt sich eine Textdatei erzeugen, die *nicht* mit einem Zeilenwechsel endet. Eine solche Textdatei verursacht die gleichen Probleme wie eine Binärdatei.

wird schon bei zwei Streams reichlich unübersichtlich und ist zudem bei jeder Art von Stream-I/O in gleicher Form zu wiederholen.

Sprachelement für zuverlässige `close`-Aufrufe

Java 7 bietet mit der **automatischen Ressourcenverwaltung** (*Automatic Resource Management*, ARM) ein kompaktes, elegantes Sprachmittel, das sich zuverlässig um diese Probleme kümmert. Die Konstruktion, die auch als *try with resource* bezeichnet wird, hat die folgende Form:

```
try(definition; ...; definition) {
    statement ...
}
```

Schließen von Ressourcen

Im Kopf werden eine oder mehrere „Ressourcen“ definiert und stehen im Rumpf zur Verfügung. Beim Verlassen der Kontrollstruktur werden automatisch die `close`-Methoden aller im Kopf definierten Ressourcen aufgerufen.[31] Auch wenn eine oder mehrere der `close`-Aufrufe scheitern, werden die anderen ausgeführt.

Anders als andere `try`-Blöcke kann ein ARM-Block syntaktisch alleine stehen und braucht kein nachfolgendes `catch` oder `finally`. Wenn Exceptions behandelt werden sollen, geschieht das wie üblich durch nachfolgende `catch`-Blöcke.

Mehrfache Exceptions

Beim Verlassen eines ARM-Blocks können mehrere Exceptions auflaufen:[32] Wenn eine Methode im Rumpf eine Exception auslöst und daraufhin auch noch einer oder mehrere `close`-Aufrufe scheitern, sammeln sich alle diese Exceptions an. Nach außen dringt zunächst nur die letzte Exception. Die `Throwable`-Methode `getSuppressed()` liefert in diesem Fall ein Array mit allen vorangegangenen Exceptions, sodass keine Information über Fehler verloren geht.

Listen von Ressourcen

Die Variablendefinitionen im Kopf sind mit Strichpunkten getrennt.[33] Diese Variablen sind automatisch `final` und *müssen sofort* mit Ressourcen initialisiert werden. Sie gelten ab der Definition bis zum Ende des Rumpfes. Nachfolgende Ressourcen können sich auf vorhergehende Ressourcen beziehen, aber nicht umgekehrt.

Das Programm `StreamCopyARM` entspricht `StreamCopyClosing` (Listing 1.8) und zeigt den Einsatz dieses Sprachmittels:

```
import java.io.*;

public class StreamCopyARM {
    public static void main(String... args) throws IOException {
```

[31] Die `close`-Aufrufe erfolgen in der umgekehrten Definitionsreihenfolge, also von der letzten bis zur ersten im Kopf genannten Ressource.

[32] Bis zur Java-Version 6 war das nicht möglich.

[33] Nach der letzten (oder einzigen) Definition folgt kein Strichpunkt.

```
        try(InputStream input = System.in;
            OutputStream output = System.out) {
            int code = input.read();
            while(code >= 0) {
                output.write(code);
                code = input.read();
            }
        }
    }
}
```

Listing 1.10: Zeichenweises Kopieren von Streams unter Kontrolle des ARM.

Die expliziten `close`-Aufrufe von `StreamCopyClosing` (Listing 1.8) übernimmt hier das ARM. Sie werden automatisch beim Verlassen des `try`-Blocks ausgeführt. Weiter werden immer beide Streams geschlossen, selbst wenn einer der `close`-Aufrufe scheitert.

ARM-Blöcke erweitern die Ausdrucksmächtigkeit von Java nicht, sondern erlauben nur eine einfachere Schreibweise für eine ansonsten komplexe Konstruktion. Der Compiler expandiert ARM-Blöcke beim Übersetzen in geschachtelte `try/catch`-Blöcke.

Interfaces `AutoCloseable` und `Closeable`

`AutoCloseable` als Typ von Ressourcen

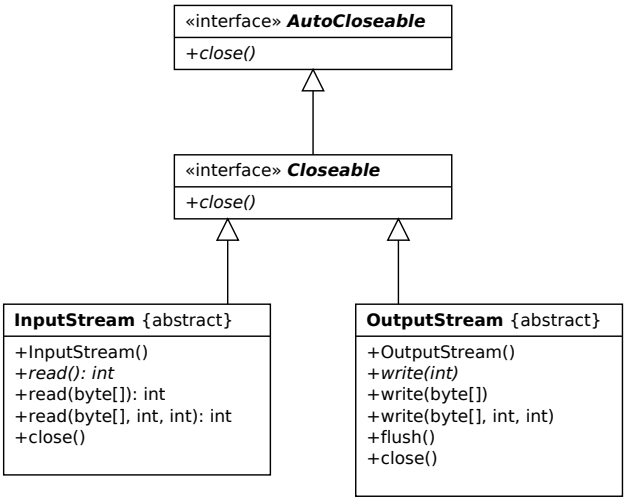
Das ARM ist nicht auf Streams beschränkt, sondern allgemeiner einsetzbar. Jede Klasse, die das Interface `java.lang.AutoCloseable` mit der einzigen Methode

```
void  close() throws Exception
```

implementiert, kann der Kontrolle des ARM ungeordnet werden. Von `AutoCloseable` ist das speziellere Interface `java.io.Closeable` mit einer strengeren Fassung von `close` abgeleitet. Während `AutoCloseable` weiter keine Anforderungen an die `close`-Implementierung stellt, verlangt `Closeable` eine *idempotente* Methode. **Idempotenz** bedeutet, dass ein einziger Aufruf die gleiche Wirkung hat wie beliebig viele aufeinanderfolgende Aufrufe. Anders ausgedrückt: Ein `Closeable`-Objekt darf beliebig oft nacheinander geschlossen werden, ein `AutoCloseable`-Objekt nur ein einziges Mal.

Idempotentes `close` in `Closeable`

Die verschiedenen I/O-Klassen im Package `java.io` implementieren alle das Interface `Closeable`. Das folgende Diagramm veranschaulicht den Zusammenhang:



1.3 File-I/O

1.3.1 Konkrete Datenquellen und -senken

`Stream`-Klassen für konkrete Quellen und Senken

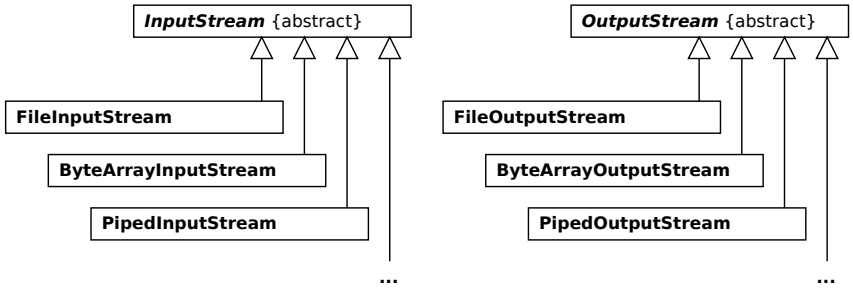
Die Basisklassen `InputStream` und `OutputStream` repräsentieren keine bestimmten Datenquellen beziehungsweise -senken, sondern stecken nur die minimale Funktionalität aller Streams ab. Bisher wurden in diesem Kapitel nur die drei Standard-I/O-Objekte als konkrete, zu den ABCs kompatible Objekte verwendet. So nützlich die Standardein- und -ausgabe auch sein mag, gibt es doch viele weitere interessante Kommunikationspartner. Die folgende Liste zeigt I/O-Klassen für einige andere konkrete Quellen und Senken:

`FileInputStream`, `FileOutputStream`
: File im Filesystem.

`ByteArrayInputStream`, `ByteArrayOutputStream`
: Byte-Array im eigenen Programm.

`PipedInputStream`, `PipedOutputStream`
: Anderer Thread (siehe Kapitel 6, Seite 361).

Alle diese Klassen sind von den ABCs abgeleitet und definieren die entsprechenden Methoden. Sie werden auf die gleiche Weise verwendet wie in den bisher gezeigten Beispielprogrammen.

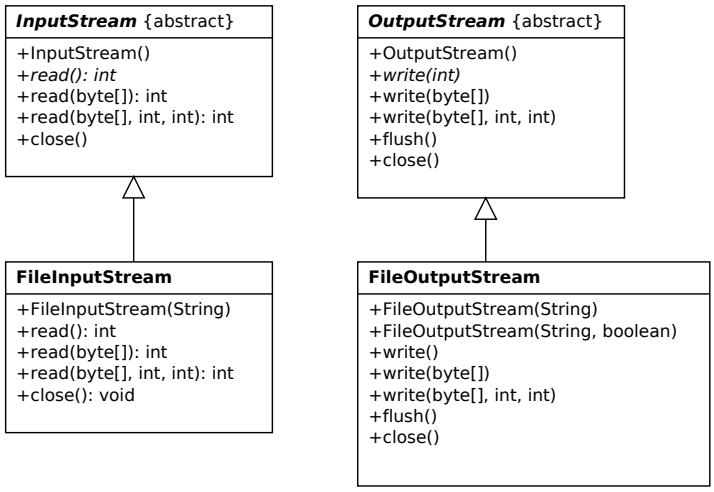


Es gibt auch ausgefallenere Stream-Klassen, wie zum Beispiel `ObjectOutputStream` zum Schreiben ganzer Objekte (siehe Kapitel 2, Seite 125) oder `AudioInputStream` zum Lesen von Klangdateien.[34]

1.3.2 Filestreams

Klassen für Dateiein- und -ausgabe

Die beiden konkreten I/O-Klassen `FileInputStream` und `FileOutputStream` repräsentieren Dateien und zählen zu den häufiger verwendeten konkreten I/O-Klassen.



Konstruktoren

Dateinamen im Konstruktor

Ihre Konstruktoren erwarten die Angabe einer bestimmten Quelldatei beziehungs-

[34] Der Umgang mit Audiodaten wird in diesem Buch nicht behandelt.

weise Zieldatei als String. Hier zwei Beispiele:[35]

```
new FileInputStream("output.txt")
new FileInputStream("Hello.java")
```

Im `FileInputStream`-Konstruktor wird sichergestellt, dass die genannte Datei tatsächlich existiert. Andernfalls wird eine `FileNotFoundException` geworfen, ein von `IOException` abgeleiteter Exceptiontyp:

Fehler beim Dateizugriff

```
import java.io.*;

public class FileStreamCtors {
    public static void main(String... args) throws IOException {
        try(InputStream input = new FileInputStream(args[0]);
            OutputStream output = new FileOutputStream(args[1])) {
        }
        catch(FileNotFoundException fnfx) {
            System.out.println("Houston, we have a problem!");
            throw fnfx;
        }
    }
}
```

Listing 1.11: Öffnen von Files, die möglicherweise nicht existieren, zum Lesen und Schreiben.

Dieses Programm erwartet zwei Kommandozeilenargumente, den Namen einer Datei zum Lesen und den Namen einer Datei, die geschrieben werden soll. Vorsicht! Die Ausgabedatei, hier `newfile`, wird kommentarlos überschrieben![36]

```
$ java FileStreamCtors FileStreamCtors.java newfile
```

Das Programm bricht mit einer Exception ab, wenn die Eingabedatei nicht existiert, wie im folgenden Beispiel `abrakadabra`:

```
$ java FileStreamCtors abrakadabra newfile
Houston, we have a problem!
Exception ... java.io.FileNotFoundException: abrakadabra (No such file or directory)
```

Die gleiche `FileNotFoundException` wird allerdings auch dann geworfen, wenn die Datei zwar existiert, aber andere Hindernisse den Zugriff blockieren, wie zum Beispiel mangelnde Zugriffsrechte. Im folgenden Beispiel wird angenommen, dass der Benutzer die Datei `topsecret` überhaupt nicht öffnen darf.

Ursachen von Zugriffsfehlern

[35] Der Backslash als Trenner zwischen Pfadelementen in Windows muss im Quelltext entwertet werden.

[36] Vorausgesetzt der Benutzer verfügt überhaupt über die Berechtigung zum Schreiben von `newfile`.

```
$ java FileStreamCtors topsecret newfile
Houston, we have a problem!
Exception ... java.io.FileNotFoundException: topsecret (Permission denied)
```

Obwohl der Exceptiontyp (`FileNotFoundException`) hier etwas irreführend ist, benennt der Message-String („Permission denied“) die wahre Ursache des Problems.

Entsprechendes gilt für den `FileOutputStream`-Konstruktor: Er verlangt zwar nicht, dass die Zieldatei existiert, legt sie aber neu an und stellt dabei sicher, dass das auch gelingt. Wieder sind dazu unter anderem ausreichende Berechtigungen nötig.

```
$ java FileStreamCtors FileStreamCtors.java topsecret
Houston, we have a problem!
Exception ... java.io.FileNotFoundException: topsecret (Permission denied)
```

Betriebssystem-Abhängigkeiten

Betriebssystem-Abhängigkeit von Dateinamen

Aus der Sicht der Konstruktoren sind die angegebenen Pfadnamen beliebige Strings. Allerdings werden sie auf Filenamen des zugrunde liegenden Betriebssystems abgebildet, das keineswegs beliebige Java-Strings akzeptiert. Daher kommen an dieser Stelle Systemabhängigkeiten ins Spiel. Beispielsweise erlaubt Windows keine Doppelpunkte in Filenamen, während Unix das akzeptiert:

```
$ java FileStreamCtors FileStreamCtors.java unix:only
$
```

Auf einem Windows-Rechner dagegen:

```
C:\> java FileStreamCtors FileStreamCtors.java unix:only
Houston, we have a problem!
Exception ... java.io.FileNotFoundException: unix:only (Invalid argument)
```

Erneut gibt der Message-String der Exception („Invalid argument“) den Hinweis auf die tatsächliche Ursache des Problems.[37]

Angabe von Pfadnamen

Die File-I/O-Konstruktoren akzeptieren nicht nur Filenamen, sondern komplette Pfadnamen. Wie die Namen selbst sind auch die Pfadangaben systemspezifisch. In Unix dient zum Beispiel der Schrägstrich / als Pfadtrenner:

```
$ java FileStreamCtors ./FileStreamCtors.java /tmp/output
$
```

[37] In FAT-Filesystemen sind aus historischen Gründen einige Filenamen verboten, wie zum Beispiel `aux`, `con` und `lpt1`. Das gilt auch dann, wenn eine Extension angefügt wird, wie zum Beispiel `aux.txt`.

Windows kennt zusätzlich Laufwerksbezeichnungen und benutzt das Backslash-Zeichen (\) als Pfadtrenner.[38]

```
c:\> java FileStreamCtors .\FileStreamCtors.java c:\windows\Temp\output
c:\>
```

Pfadtrenner

Portable Pfadangaben

Ein Java-Programm, das sich explizit auf Windows- oder Unix-Pfadangaben mit \ oder / als Trenner bezieht, ist nicht portabel. Es kommt damit auch dem Anspruch von Java auf Systemneutralität nicht nach.

In der Klasse `java.io.File` sind vier Konstanten definiert, die dieses Problem lösen:

`public static final String separator;`
: Ein String mit dem Trennzeichen für Pfadelemente auf dem gerade laufenden System (`"\"` auf Windows und `"/"` auf Unix).

`public static final String pathSeparator;`
: Ein String mit dem Trennzeichen für Listen von Pfaden auf dem gerade laufenden System (`";"` auf Windows und `":"` auf Unix).[39]

Diese beiden Konstanten werden durch zwei `char`-Konstanten `separatorChar` und `pathSeparatorChar` ergänzt, die die betreffenden Zeichen einzeln zur Verfügung stellen.

Das folgende Programm öffnet eine Datei, deren Directories und Name einzeln auf der Kommandozeile angegeben sind. Es funktioniert auf allen Systemen, sofern die Pfadelemente zulässig sind:

```
import java.io.*;

public class DirFile {
    public static void main(String... args) throws IOException {
        String pathname = "";
        for(String arg: args)
            pathname += File.separator + arg;
        try(InputStream input = new FileInputStream(pathname.substring(1))) {
            // ...
```

[38] Java auf Windows übersetzt stillschweigend den Unix-Pfadtrenner (/) in die Windows-eigene Notation. Das gilt allerdings nur für die JVM und nicht für andere Programme.

[39] Dieses Zeichen trennt beispielsweise die Liste der Suchpfade in der Umgebungsvariablen `CLASSPATH`.

```
        }
    }
}
```

Listing 1.12: Lesen eines Files mit gegebenem Directory und Namen.

Verlängern von Dateien

Verlängern von Dateien

Die Klasse `FileOutputStream` definiert zwei überladene Konstruktoren:

```
FileOutputStream(String filename)
FileOutputStream(String filename, boolean append)
```

Der zweite Konstruktor akzeptiert das Flag `append`, das über den Umgang mit einer bereits existierenden Datei entscheidet. Mit `append = false` wird der Inhalt der vorhandenen Datei gelöscht, ebenso wie beim Aufruf des ersten Konstruktors.[40] Bei `append = true` bleibt der Inhalt dagegen erhalten und wird verlängert. Wenn die Ausgabedatei nicht existiert, verhalten sich die Konstruktoren gleich und legen die Datei neu an.

Das folgende Programm `FileCopy` akzeptiert zwei Filenamen auf der Kommandozeile. Es kopiert den Inhalt der erstgenannten Datei in die zweitgenannte. Wenn ein beliebiges drittes Kommandozeilenargument angefügt wird, dann wird die Zieldatei nicht ersetzt, sondern verlängert:

```
import java.io.*;

public class FileCopy {
    public static void main(String... args) throws IOException {
        boolean append = args.length > 2;
        try(InputStream input = new FileInputStream(args[0]);
            OutputStream output = new FileOutputStream(args[1], append)) {
            for(int code = input.read(); code >= 0; code = input.read())
                output.write(code);
        }
    }
}
```

Listing 1.13: Zeichenweises Kopieren eines Files in ein anderes.

Dieses Programm ist eine fast unveränderte Kopie von `StreamCopyARM` (Listing 1.10). Lediglich die Initialisierung der Streams wurde ausgetauscht, aber der Rest nicht angetastet.

[40] Tatsächlich wird nicht die ganze Datei zuerst gelöscht und dann völlig neu angelegt, sondern nur der *Inhalt* der bestehenden Datei entleert. Die Datei selbst bleibt erhalten, wie man in Unix beispielsweise an der *changed*-Zeitmarke oder an der *inode*-Nummer erkennen kann.

Das folgende Beispiel demonstriert, dass das Programm funktioniert. Zuerst wird der eigene Bytecode in ein anderes Directory kopiert. Dann wird die Kopie dort neu gestartet, die wiederum klaglos von der JVM akzeptiert wird:

```
$ java FileCopy FileCopy.class /tmp/FileCopy.class
$ cd /tmp
$ java FileCopy FileCopy.class /dev/null
```

Die nächsten Aufrufe fügen den Quelltext mehrmals hintereinander an dieselbe Zieldatei an:

```
$ java FileCopy FileCopy.java /tmp/FileCopy.java
$ java FileCopy FileCopy.java /tmp/FileCopy.java append
$ java FileCopy FileCopy.java /tmp/FileCopy.java append
```

1.3.3 Blockweise Übertragung

Performance-Problem bei Übertragung einzelner Bytes

Das Programm `FileCopy` (Listing 1.13) erfüllt zwar seine Aufgabe, allerdings zeigt sich beim Kopieren größerer Dateien eine auffallende Ineffizienz. Um dieses Problem genauer zu untersuchen, gibt das Programm `TimedFileCopy` auch Laufzeit und Durchsatz aus:

```
import java.io.*;
import static java.lang.System.*;

public class TimedFileCopy {
    public static void main(String... args) throws IOException {
        long tally = 0;
        final long start = currentTimeMillis();

        try(InputStream input = new FileInputStream(args[0]);
            OutputStream output = new FileOutputStream(args[1])) {
            for(int code = input.read(); code >= 0; code = input.read()) {
                output.write(code);
                tally++;
            }
        }

        final long elapsed = currentTimeMillis() - start;
        out.printf("Time: %d ms%n", elapsed);
        out.printf("Rate: %d bytes/sec%n", tally*1000/elapsed);
    }
}
```

Listing 1.14: Zeichenweises Kopieren eines Files mit Laufzeitmessung.

Auf dem System des Autors zeigt sich eine ziemlich vernichtende Performance beim Kopieren einer Datei von 10 Megabyte Länge:[41]

```
$ java TimedFileCopy 10meg copyof10meg
Time: 35968 ms
Rate: 291530 bytes/sec
```

Dieser Durchsatz ist für ernsthafte Anwendungen bei Weitem zu niedrig. Der Grund liegt in der hohen Anzahl verhältnismäßig kostspieliger `read`- und `write`-Aufrufe. Daran ändert auch die Pufferung des Betriebssystems nichts, denn alleine die Abwicklung von `read` und `write` innerhalb der JVM ist eine teure Angelegenheit.

`read`- und `write`-Methoden mit `byte`-Arrays

Die abstrakten Basisklassen `InputStream` und `OutputStream` definieren aus diesem Grund verschiedene überladene `read`- und `write`-Methoden:

```
int read()

int read(byte[] buffer)

int read(byte[] buffer, int start, int num)

void write(int code)

void write(byte[] buffer)

void write(byte[] buffer, int start, int num)
```

Rückgabewert bei blockweisem Lesen

Die erste, parameterlose `read`-Methode wurde bereits vorgestellt. Sie liest ein Byte und liefert den Wert zurück. Die zweite `read`-Methode versucht das Array `buffer` komplett aufzufüllen und holt dabei so viele Bytes wie möglich *in einem Zug*. Sie liefert die Anzahl der tatsächlich gelesenen Bytes als Ergebnis zurück. Je nach Datenquelle kann das Ergebnis mit der Arraylänge übereinstimmen oder aber auch darunterliegen. Selbst das Ergebnis 0 ist möglich, wenn überhaupt keine Daten gelesen werden konnten. Das heißt nicht, dass die Eingabe beendet ist, sondern nur,

[41] Wie immer bei derartigen Messungen schwanken die genauen Zahlen von einem System zum anderen, aber auch bei mehreren Aufrufen auf demselben System. Für halbwegs brauchbare Zahlen sollten die gemessenen Laufzeiten im Sekundenbereich liegen. Weiter sollten wenigstens fünf Messungen durchgeführt werden, von denen die beiden Randwerte ignoriert und aus den übrigen der Mittelwert gebildet wird.

dass *im Moment* keine Daten zur Verfügung stehen. Die letzte `read`-Methode arbeitet fast genauso wie die zweite, versucht aber nur einen Ausschnitt des Arrays zu füllen. Die beiden folgenden Aufrufe sind gleichwertig:

```
read(buffer)
read(buffer, 0, buffer.length)
```

Ende der Eingabe bei blockweisem Lesen

Obwohl alle drei `read`-Methoden ein `int`-Ergebnis liefern, hat das Ergebnis doch eine unterschiedliche Bedeutung: Die parameterlose Methode liefert das gelesene Byte selbst, die beiden anderen die *Anzahl* der in das Array übertragenen Bytes. In allen Fällen zeigt -1 das Eingabeende an. Weitere `read`-Aufrufe sind dann nicht mehr zulässig, sie könnten ohnehin keine Daten mehr liefern.

Entsprechend arbeiten die `write`-Methoden: Die erste gibt ein Byte aus. Die zweite gibt alle Bytes des Arrays auf einmal aus. Die dritte `write`-Methode gibt den gewünschten Abschnitt des Arrays aus. Wiederum sind die folgenden Aufrufe äquivalent:

```
write(buffer)
write(buffer, 0, buffer.length)
```

Die `write`-Methoden blockieren alle so lange, bis die übergebenen Daten ausgegeben sind. Im Fehlerfall werfen sie eine `IOException`.

Effizientes Kopieren von Streams

Mit den überladenen Methoden lässt sich eine schnellere Fassung von `TimedFileCopy` (Listing 1.14) schreiben:

```
import java.io.*;
import static java.lang.System.*;

public class BlockedFileCopy {
    public static void main(String... args) throws IOException {
        long tally = 0;
        long start = currentTimeMillis();
        boolean append = args.length > 2;
        try(InputStream input = new FileInputStream(args[0]);
            OutputStream output = new FileOutputStream(args[1], append)) {
            byte[] buffer = new byte[1 << 16];
            for(int count = input.read(buffer); count >= 0; count = input.read(buffer)) {
                tally += count;
                output.write(buffer, 0, count);
            }
        }

        long elapsed = currentTimeMillis() - start;
        out.printf("Time: %d ms%n", elapsed);
        out.printf("Rate: %d bytes/sec%n", tally*1000/elapsed);
    }
```

```
}
```

Listing 1.15: Blockweises Kopieren eines Files mit Laufzeitmessung.

Vom Himmel fällt hier die Wahl der Arraylänge von 2^{16} Bytes = 64 kB. Tatsächlich gibt es keine einfache Rechnung, mit der die ideale Länge gezielt bestimmt werden könnte. Die Erfahrung zeigt jedoch, dass ein Wert in der hier verwendeten Größenordnung auf typischen Desktop-Systemen gut funktioniert.

Die Umstellung auf die blockweise Ein- und Ausgabe wirkt sich drastisch auf die Laufzeit aus:

```
$ java BlockedFileCopy 10meg copyof10meg
Time: 86 ms
Rate: 121927441 bytes/sec
```

Bei längeren Files ergibt sich eine Gesamtlaufzeit in einer aussagekräftigeren Größenordnung:[42]

```
$ java BlockedFileCopy 1gig copyof1gig
Time: 1295 ms
Rate: 829144265 bytes/sec
```

Wahl einer passenden Blockgröße

Die folgende Programmvariante erwartet auf der Kommandozeile als zusätzliches drittes Argument den Exponenten der Zweierpotenz, die als Arraygröße verwendet wird. Damit lassen sich die Auswirkungen verschiedener Blockgrößen experimentell untersuchen:

```
import java.io.*;
import static java.lang.System.*;

public class VariableBlockFileCopy {
    public static void main(String... args) throws IOException {
        long tally = 0;
        long start = currentTimeMillis();
        int size = 1 << Integer.parseInt(args[2]);

        try(InputStream input = new FileInputStream(args[0]);
            OutputStream output = new FileOutputStream(args[1])) {
            byte[] buffer = new byte[size];
            for(int count = input.read(buffer); count >= 0; count = input.read(buffer)) {
                tally += count;
                output.write(buffer, 0, count);
            }
        }
```

[42] Diese Werte wurden auf einer Ramdisk gemessen, auf der Latenzen eines Datenträgers keine Rolle spielen. Lediglich die Verwaltung des Filesystems selbst wirkt sich noch aus.

```
            long elapsed = currentTimeMillis() - start;
            out.printf("Time: %d ms%n", elapsed);
            out.printf("Rate: %d bytes/sec%n", tally*1000/elapsed);
        }
    }
```

Listing 1.16: Blockweises Kopieren eines Files mit Laufzeitmessung und wählbarer Blockgröße.

Die folgende Tabelle zeigt den Datendurchsatz in Abhängigkeit von der Arraygröße. Die abgedruckten Zahlen haben natürlich nur für das gerade verwendete System eine Bedeutung und sind auf anderen Rechnern gegenstandslos. Dennoch sind die Verhältnisse interessant.

64 Bytes	2^6	23 MB/s
256 Bytes	2^8	88
1 kB	2^{10}	287
4 kB	2^{12}	680
16 kB	2^{14}	769
32 kB	2^{15}	792
64 kB	2^{16}	829
128 kB	2^{17}	858
256 kB	2^{18}	574
1 MB	2^{20}	292
4 MB	2^{22}	228

Ab einer gewissen Größe macht die Verwaltung des Byte-Arrays, insbesondere die Allozierung und Initialisierung, den Gewinn durch das blockweise Lesen und Schreiben zunichte.

1.3.4 Abstrakte und konkrete Methoden

Definition neuer `Stream`-Klassen

In den abstrakten Basisklassen `InputStream` und `OutputStream` sind nur die parameterlosen Methoden `read()` und `write(int)` abstrakt definiert, die überladenen Fassungen mit Parametern dagegen konkret implementiert.

```
abstract int read()
```

```
int read(byte[] buffer)

int read(byte[] buffer, int start, int num)

abstract void write(int code)

void write(byte[] buffer)

void write(byte[] buffer, int start, int num)
```

Ein Blick in den Quelltext der Bibliotheksklassen[43] zeigt, dass die konkreten Methoden lediglich in Schleifen fortwährend die abstrakten Methoden aufrufen, bis genug Bytes verarbeitet sind, ein Fehler auftritt oder das Ende erreicht ist.

Die folgende konkrete Klasse `RandomInputStream` liefert Zufallsbytes. Dieses Beispiel zeigt, wie einfach sich eine funktionsfähige I/O-Klasse neu erstellen lässt:

```
import java.io.*;

public class RandomInputStream extends InputStream {
    public int read() {
        return (int)(Math.random()*256);
    }
}
```

Listing 1.17: Konkreter `InputStream`, der zufällige Bytes liefert.

Effiziente Implementierung von `Stream`-Klassen

Die konkreten Methoden in den ABCs sind nur Übergangslösungen. Sie sparen zunächst Aufwand bei der Implementierung einer neuen I/O-Klasse, sind aber für einen ernsthaften Einsatz zu langsam.

Konkrete, von den abstrakten Basisklassen abgeleitete I/O-Klassen sind angehalten, zusätzlich zu den abstrakten Methoden auch die anderen Methoden effizient zu redefinieren. Im Fall der File-I/O-Klassen ist das offenbar geschehen, sonst ergäben sich nicht die oben beschriebenen drastischen Effizienzunterschiede. Beim `RandomInputStream` ist das allerdings nicht unbedingt nötig, weil das Erzeugen der Zufallszahlen den Löwenanteil des Aufwands beansprucht.

[43] Der Quelltext der Laufzeitbibliothek findet sich im File `src.zip`, das in der Regel mit dem JDK installiert wird. Bei knappen Platzverhältnissen kann diese Datei ohne Schaden gelöscht werden. Die Laufzeitbibliothek selbst ist in `rt.jar` enthalten, die natürlich essenziell ist.

1.4 Transformationen mit Filterklassen

Ein- und Ausgabe mit Manipulation der Daten

Daten können bei der Ein- und Ausgabe auf vielfältige Art manipuliert werden. Beispielsweise kann es sinnvoll sein, Daten bei der Ausgabe zu komprimieren und bei der Eingabe wieder zu expandieren, um Speicherplatz oder Übertragungsbandbreite einzusparen. Vielleicht möchte man aus Sicherheitsgründen Daten bei der Ausgabe verschlüsseln und bei der Eingabe wieder entschlüsseln.

Transformationen, wie Kompression und Verschlüsselung, sind unabhängig von einer konkreten Datenquelle oder -senke. Sie sind gleichermaßen interessant bei der Ausgabe auf Dateien, über Netzwerkverbindungen, zu anderen Programmen oder auf die Standardausgabe. Man könnte Vererbung einsetzen, um reguläre I/O-Klassen um verschiedene Transformationen zu erweitern.

Inflation von Klassen mit kombinierten Eigenschaften

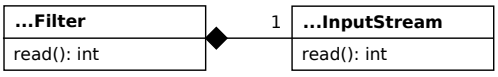
Zu jeder konkreten `OutputStream`-Klasse müsste man eine abgeleitete `Compressing`-Variante anbieten, also

```
CompressingFileOutputStream
CompressingPipedOutputStream
CompressingByteArrayOutputStream
CompressingSystemOut
```

und so weiter. Entsprechend bräuchte man zur verschlüsselten Ausgabe einen weiteren Satz abgeleiteter `Encrypting`-Klassen. Um Verschlüsselung mit Kompression zu kombinieren, wären dann noch `CompressingEncrypting`-Klassen nötig. Es ist leicht zu sehen, dass dieses Vorgehen zu einer Schwemme von Klassen führen würde. Dies ist also kein gangbarer Weg.

Filterklassen für isolierte Manipulationen

Die I/O-Filter von Java bieten eine einfachere Lösung für das Problem: Transformationen (Kompression, Verschlüsselung und so weiter) werden nicht in abgeleiteten Klassen implementiert, sondern als selbstständige „Filterklassen“. Objekte der Filterklassen werden zur Laufzeit an zugrunde liegende Objekte anderer I/O-Klassen gekoppelt, deren Daten sie durchreichen und dabei manipulieren. Die folgende Skizze zeigt einen Eingabefilter, der sich von einem `InputStream` „ernährt“:



Zu den bereits verfügbaren konreten I/O-Klassen kommt also eine Auswahl von Filterklassen hinzu, von denen jede eine bestimmte Transformation implementiert. Jede konkrete I/O-Klasse kann dann mit jeder Filterklasse[44] kombiniert werden.

[44] Natürlich können nur Ausgabe-Byteströme mit Ausgabefiltern und Eingabe-Byteströme mit Eingabefiltern verbunden werden.

1.4.1 Ableitung und Komposition

Filterstreams vs. konkrete Streams

Es gibt damit zwei Sorten von I/O-Klassen:

- Die konkreten I/O-Klassen repräsentieren eine bestimmte Datenquelle oder -senke.[45] Sie sind einzeln nutzbar. Die einheitliche Handhabung stellen die abstrakten Basisklassen `InputStream` und `OutputStream` sicher.
- Objekte der **Filterklassen** sind dagegen auf eine Quelle oder Senke angewiesen. Sie sind alleine nicht „lebensfähig“, sondern brauchen immer ein zweites I/O-Objekt als Lieferanten oder Abnehmer.

Filter *sind* Streams und *nutzen* Streams

Auch Filterklassen sind von `InputStream` und `OutputStream` abgeleitet und stellen damit *dieselben* Methoden zur Verfügung wie konkrete I/O-Klassen. Anwendungen müssen also nicht wissen, ob sie direkt mit einem konkreten I/O-Objekt arbeiten oder ob ein Filter davorgeschaltet ist. Der Unterschied ist nur dem Konstruktoraufruf anzusehen: Ein Objekt einer konkreten I/O-Klassen wird direkt erzeugt, wie zum Beispiel

```
new FileOutputStream(filename)
```

Ein Objekt einer Filterklasse braucht ein bereits vorher existierendes I/O-Objekt. Das folgende Beispiel erzeugt zuerst einen `FileOutputStream` und setzt dann davor einen Filter, der Daten verschlüsselt.[46]

```
new EncryptingFilter(new FileOutputStream(filename))
```

An den beiden `new`-Aufrufen ist deutlich zu erkennen, dass hier zwei Objekte im Spiel sind.

Der Verschlüsselungsfilter arbeitet auch mit jedem anderen `OutputStream` als Abnehmer zusammen:

```
new EncryptingFilter(new ByteArrayOutputStream(...))
new EncryptingFilter(new PipedOutputStream(...))
```

Filterketten

Konkatenation von Filtern

Das Potenzial von Filtern liegt darin, dass sie selbst `InputStreams` beziehungsweise

[45] Der Begriff „konkret“ hat hier nichts mit Typen zu tun. Er drückt aus, dass das betreffende I/O-Objekt über eine eigene Datenquelle oder -senke verfügt und auf kein anderes I/O-Objekt angewiesen ist.

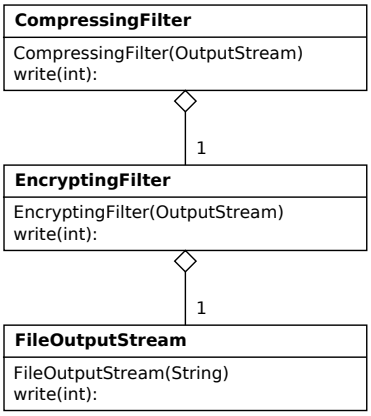
[46] `EncryptingFilter` ist ein hypothetischer Name. In der Laufzeitbibliothek stehen Filterklassen zur Verschlüsselung zur Verfügung, deren Initialisierung allerdings etwas komplizierter ist. Einzelheiten finden Sie auf Seite 485.

`OutputStreams` sind. Ein Filter kann also nicht nur mit einem konkreten I/O-Objekt verbunden werden, sondern ebenso mit einem anderen Filter! Daraus entstehen Filter*ketten*, die aber immer mit einem konkreten I/O-Objekt enden. Jeder Filter in einer Kette manipuliert die durchfließenden Daten auf seine Art.

Filter als Baukasten

Eine Anwendung kann aus einem ganzen Angebot von Filtern und konkreten I/O-Klassen beliebige Ketten zusammenstellen. Im folgenden Beispiel wird eine Kette aus zwei Ausgabefiltern und einem konkreten Ausgabe-Bytestrom aufgebaut. Zur besseren Übersichtlichkeit werden drei einzelne Anweisungen verwendet. Die Variable `compressedAndEncrypted` enthält am Ende den vordersten Filter der Kette, der als Eingang dient.

```
OutputStream file = new FileOutputStream(filename);
OutputStream encrypted = new EncryptingFilter(file);
OutputStream compressedAndEncrypted = new CompressingFilter(encrypted);
```



Auch eine einzige Anweisung reicht aus, um die Konstruktion aufzubauen:

```
OutputStream compressedAndEncrypted =
                        new CompressingFilter(
                            new EncryptingFilter(
                                new FileOutputStream(filename)));
```

Bytes, die auf `compressedAndEncrypted` ausgegeben werden, durchlaufen die Kette Stufe um Stufe: Zuerst werden sie vom `CompressingFilter` komprimiert, dann die komprimierten Daten vom `EncryptingFilter` verschlüsselt und schließlich die komprimierten und verschlüsselten Daten vom `FileOutputStream` auf eine Datei geschrieben.[47] Filterklassen spielen eine interessante Zwitterrolle:

[47] In diesem Beispiel könnten die beiden Filter auch getauscht werden. Das wäre aber nicht sinn-

- Zum einen sind sie von einer der ABCs *abgeleitet* und bieten damit für den Anwender die gleiche Schnittstelle wie alle anderen I/O-Klassen.
- Zum anderen *referenzieren* Sie eine andere I/O-Klasse, von der sie sich „ernähren“ beziehungsweise auf die sie „abladen“.

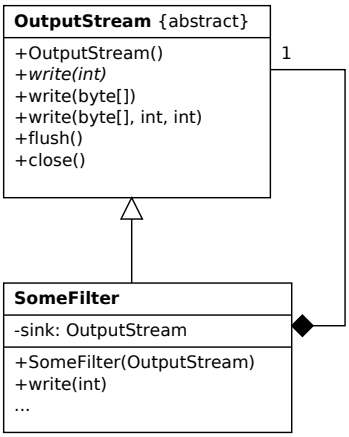
Implementierung eines Filters

Der Code einer hypothetischen Filterklasse `SomeFilter` sieht also etwa folgendermaßen aus:

```
import java.io.*;

public class SomeFilter extends OutputStream {
    private final OutputStream sink;

    SomeFilter(final OutputStream sink) {
        this.sink = sink;
    }

    public void write(int code) throws IOException {
        // Manipulation
        sink.write(code);
    }
}
```

Listing 1.18: Rahmenklasse für einen Ausgabefilter.

Im Code sind die zwei Bezüge auf die ABC `OutputStream` zu sehen: Einerseits taucht `OutputStream` als Basisklasse auf (Ableitung), andererseits als Typ der referenzierten Objektvariablen `sink` (Komposition).



voll, weil Verschlüsselung in der Regel Regelmäßigkeiten verwischt und Daten ohne Regelmäßigkeit kaum komprimiert werden können.

Die folgende Klasse `OddByteInputStream` definiert einen Eingabefilter, der nur Bytes mit ungeraden Werten durchlässt und die anderen herausfiltert:

```
import java.io.*;

public class OddByteInputStream extends InputStream {
    private final InputStream source;

    public OddByteInputStream(InputStream source) {
        this.source = source;
    }

    public int read() throws IOException {
        int code;
        do
            code = source.read();
        while(code%2 == 0);
        return code;
    }
}
```

Listing 1.19: Eingabestrom, der die ungeraden Bytes eines anderen Eingabestroms liefert.

Die folgende Anwendung erzeugt ein File mit einem Megabyte (2^{20} Bytes) ungeraden Zufallsbytes:

```
import java.io.*;

public class Write1MegOddBytes {
    public static void main(String... args) throws IOException {
        try(InputStream input = new RandomInputStream();
            InputStream oddInput = new OddByteInputStream(input);
            OutputStream output = new FileOutputStream(args[0])) {
            byte[] buffer = new byte[1 << 20];
            int tally = 0;
            while(tally < buffer.length) {
                int got = oddInput.read(buffer);
                tally += got;
                output.write(buffer, 0, got);
            }
        }
    }
}
```

Listing 1.20: Ausgabe von einer Million zufälligen ungeraden Bytes.

Diese Anwendung arbeitet mit blockweiser Ein- und Ausgabe. Weder die konkrete Input-Klasse `RandomInputStream` (Listing 1.17) noch die Filterklasse `OddByteInputStream` (Listing 1.19) definieren eine `read`-Methode zum blockweisen Lesen. Beide erben

diese Methode von derselben Basisklasse `InputStream`. Die dort definierte Default-Implementierung ruft in einer Schleife wiederum die parameterlose `read`-Methode der konkreten Klassen auf. In die Zieldatei wird die gewünschte Anzahl Bytes geschrieben. Der Filter muss aber eine größere Anzahl Bytes von seiner Quelle, dem `RandomInputStream`, holen, weil er die geraden Bytes verwirft.

Ein etwas aufwendigerer Ausgabefilter `CollapseWhitespace` zieht allen Zwischenraum, der Java-Identifier trennt, zu einem einzigen Leerzeichen zusammen. Der Algorithmus im Einzelnen ist an dieser Stelle nicht wichtig. Diese Klasse soll das Konstruktionsschema eines Filters veranschaulichen:

```
import java.io.*;

public class CollapseWhitespace extends OutputStream {
    private final OutputStream sink;

    private boolean lastPrintingWasIdchar = false;  // Identifier-Zeichen vor Blanks?

    private boolean lastWasSpace = true;        // Blank direkt davor?

    public CollapseWhitespace(OutputStream sink) {
        this.sink = sink;
    }

    public void write(int code) throws IOException {
        boolean currentIsSpace = Character.isWhitespace(code);
        if(!currentIsSpace) {
            boolean currentIsIdchar = Character.isJavaIdentifierPart(code);
            if(lastWasSpace && lastPrintingWasIdchar && currentIsIdchar)
                sink.write(' ');
            sink.write(code);

            lastPrintingWasIdchar = currentIsIdchar;
        }
        lastWasSpace = currentIsSpace;
    }
}
```

Listing 1.21: Ausgabefilter, der Zwischenraum herausfiltert.

Die folgende Anwendung `StompJavaSource` arbeitet wie `ConsoleEcho` (Listing 1.2), schickt die Ausgabe aber durch einen `CollapseWhitespace`-Filter, der Zwischenraum herauskürzt:

```
import java.io.*;
import static java.lang.System.*;

public class StompJavaSource {
    public static void main(String... args) throws IOException {
        try(InputStream input = in;
```

```
            OutputStream output = new CollapseWhitespace(out)) {
            for(int code = input.read(); code >= 0; code = input.read())
                output.write(code);
        }
    }
}
```

Listing 1.22: Anwendung des Ausgabefilters, der Zwischenraum löscht.

`StompJavaSource` ruft zwar `write` mit jedem einzelnen Zeichen auf, tatsächlich ausgegeben wird aber nur ein Teil davon. Auf den eigenen Quelltext angewendet, wird eine einzige lange Zeile ohne jedes Layout produziert. Das Ergebnis entspricht formal dem ursprünglichen Quelltext,[48] ist aber praktisch unleserlich.

```
$ java StompJavaSource < StompJavaSource.java
import java.io.*;public class StompJavaSource{public static void main(String[]args...
```

1.4.2 Abstrakte Filterklassen

Gemeinsamkeiten aller Filter

Alle Filterklassen, ob vorgegeben oder selbst definiert, haben Gemeinsamkeiten: Sie beziehen sich auf ein weiteres I/O-Objekt und initialisieren dieses im Konstruktor.

Darüber hinaus zeigt sich am vorhergehenden Beispiel `StompJavaSource` ein weiteres Problem: Wenn man die Ausgabe des Programms auffängt, sollte man vielleicht unleserlichen, aber immer noch intakten Quelltext erhalten. Trotzdem übersetzt der Compiler das Ergebnis nicht:[49]

```
$ java StompJavaSource < StompJavaSource.java > /tmp/StompJavaSource.java
$ cd /tmp
$ javac StompJavaSource.java
StompJavaSource.java:1: error: reached end of file while parsing
...
```

Das Problem liegt in der Filterklasse `CollapseWhitespace`. Der ARM-Block im Hauptprogramm ruft zwar pflichtgemäß die `close`-Methode auf, die aber in `CollapseWhitespace` selbst nicht definiert ist. Vielmehr erbt `CollapseWhitespace` die `close`-Methode der abstrakten Basisklasse `OutputStream`. Diese ererbte Methode ist aber leer und tut schlicht nichts!

Schließen des zugrunde liegenden Streams

Niemand kümmert sich also um das zweite OutputStream-Objekt, den Abnehmer

[48] Dieser Filter ist sehr simpel gestrickt und erkennt beispielsweise keine `String`- und `char`-Literale. Er eignet sich daher nicht ernsthaft, um Java-Quelltext unter Erhalt der Semantik zu kürzen.

[49] Die Ausgabe muss in ein anderes Directory geschickt werden, sonst überschreibt das Programm die Originaldatei. Hier geht es aber um ein ganz anderes Problem.

von `CollapseWhitespace`. In `StompJavaSource` (Listing 1.22) ist das die Standardausgabe `System.out`. Wegen des ausbleibenden `close`-Aufrufs gehen gepufferte Daten verloren[50] und die Ausgabe ist unvollständig. Das Problem lässt sich in `StompJavaSource` (Listing 1.22) beheben, indem man die Datensenke von `CollapseWhitespace` explizit als Ressource definiert und damit dem ARM unterordnet:

```
import java.io.*;
import static java.lang.System.*;

public class StompJavaSourceComplete {
    public static void main(String... args) throws IOException {
        try(InputStream input = in;
            OutputStream output = out;
            OutputStream cw = new CollapseWhitespace(output)) {
            for(int code = input.read(); code >= 0; code = input.read())
                cw.write(code);
        }
    }
}
```

Listing 1.23: Löscht Zwischenraum aus der Standardeingabe und schreibt den Rest auf die Standardausgabe.

In dieser Fassung werden am Ende des ARM-Blocks die `close`-Methoden aller drei Ressourcen einzeln aufgerufen, insbesondere auch die von `System.out`. Die Ausgabe ist jetzt komplett, weil sich das ARM um alle drei Streams kümmert.

Diese Lösung funktioniert, kann aber verbessert werden. `CollapseWhitespace` kennt ja die eigene Datensenke und könnte selbst eine `close`-Methode definieren, die den Aufruf an die nachgeschaltete Datensenke weitergibt. Diese Lösung ist robuster, weil sich der Anwender nicht mehr um die Datensenke des Filters kümmern muss. Das gilt nicht nur für `CollapseWhitespace`, sondern überhaupt für alle Filterklassen, die damit eine weitere Gemeinsamkeit aufweisen.

Abstrakte Filterklassen als Grundlage neuer Filter

Um alle Gemeinsamkeiten zu sammeln, sind in der Laufzeitbibliothek die Basisklassen `FilterInputStream` und `FilterOutputStream` definiert. Diese Basisklassen implementieren eine Art „leere Filter“. Neue Filterklassen lassen sich bequem davon ableiten und müssen im einfachsten Fall nur noch eine `read`- beziehungsweise `write`-Methode bereitstellen. Die Basisklassen kümmern sich um die Verwaltung der zugeordneten Quelle oder Senke. Sie redefinieren die Methoden der abstrakten Basisklassen `InputStream` und `OutputStream` und delegieren Aufrufe an die zugrunde liegenden Streams. Ausnahmen sind die folgenden Methoden:

[50] Diese Beispiel ist so gewählt, dass der ausbleibende `close`-Aufruf tatsächlich zu unübersehbaren Fehlern führt. Ein Filter, der Zeilenwechsel weitergibt, würde das das Problem dagegen kaschieren, weil das Betriebssystem bei Zeilenumbrüchen in der Regel ungefragt den Puffer leert.

`int read(byte[] buffer) // FilterInputStream`
: Ruft die eigene Methode `read(buffer, 0, buffer.length)` auf.

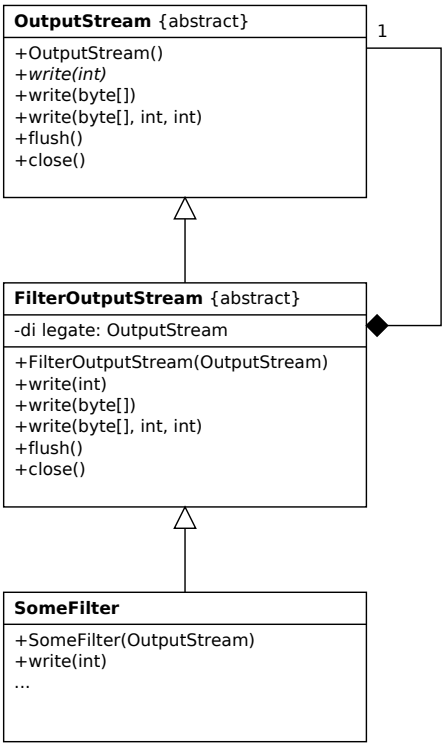
`void write(byte[] buffer) // FilterOutputStream`
: Ruft die eigene Methode `write(buffer, 0, buffer.length)` auf.

`void write(byte[] buffer, int start, int length) // FilterOutputStream`
: Ruft mit jedem Byte einzeln die Methode `void write(int buffer)` auf.

Der oben definierte hypothetische Filter `SomeFilter` (Listing 1.18) ändert sich damit wie folgt

```
import java.io.*;

public class SomeFilter extends FilterOutputStream {    // vorher OutputStream
    // keine Objektvariable

    SomeFilter(OutputStream sink) {
        super(sink);                                     // vorher this.sink = sink;
    }

    public void write(int code) throws IOException {
        // Manipulation ...
        super.write(code);                               // vorher sink.write();
    }
}
```

Das folgende Klassendiagramm zeigt die Anordnung der Basisklassen:



Vordefinierte Methoden delegieren an zugrunde liegenden Stream

Aus struktureller Sicht handelt es sich dabei um die Delegation von einem Objekt an ein anderes. Die `FilterStream`-Basisklassen bieten die gleichen Methoden wie die eingekapselten nachgeschalteten Streams an. Die `FilterStream`-Methoden rufen lediglich die entsprechenden Methoden des nachgeschalteten Streams auf. Sie reichen alle Parameter durch und liefern das Ergebnis wieder zurück. Die Klasse `FilterOutputStream` sieht damit etwa folgendermaßen aus:

```
public class FilterOutputStream extends OutputStream {
    private final OutputStream delegate;

    public FilterOutputStream(OutputStream delegate) {
        this.delegate = delegate;
    }

    public void write(int code) throws IOException {
        delegate.write(code);
    }

    public void write(byte[] code) throws IOException {
        delegate.write(code);
    }

    public void close() throws IOException {
```

```
        delegate.close();
    }

    // ... und so weiter mit allen OutputStream-Methoden
}
```

Listing 1.24: Muster der ABC für Ausgabefilter.

1.4.3 Konkrete Filterklassen

Ableitung konkreter Filterklassen

Konkrete Filterklassen erben jetzt von `FilterInputStream` oder `FilterOutputStream` statt direkt von den abstrakten Basisklassen `InputStream` und `OutputStream`. Sie erben damit einen brauchbaren Satz delegierender Methoden, darunter insbesondere `close`.

Filterklassen verwalten die angekoppelten I/O-Objekte nicht mehr selbst, sondern überlassen das den `FilterStream`-Basisklassen. Entsprechende Objektvariablen in den abgeleiteten Klassen fallen weg, der Konstruktor ruft nur noch den Basisklassenkonstruktor auf.

Die zugrunde liegenden Streams sind immer noch über die `protected`-Variablen `in` beziehungsweise `out` erreichbar. Im Allgemeinen rufen die abgeleiteten Filter aber entsprechende Methoden der Filter-Basisklasse auf und wenden sich nicht direkt an die zugrunde liegenden Streams.

Die Klasse `OddByteInputStream` (Listing 1.19) lässt sich auf der Grundlage der Basisklasse `FilterInputStream` neu definieren:

```
import java.io.*;

public class OddByteFilterInputStream extends FilterInputStream {
    public OddByteFilterInputStream(InputStream source) {
        super(source);
    }

    public int read() throws IOException {
        int code;
        do
            code = super.read();
        while(code%2 == 0);
        return code;
    }
}
```

Listing 1.25: Eingabefilter, der nur Bytes mit ungeraden Werten durchlässt.

Auch der Filter zum Kürzen von Zwischenraum in Java-Quelltext `CollapseWhitespace` (Listing 1.21) wird jetzt von der Filter-Basisklasse abgeleitet:

```
import java.io.*;

public class CollapseWhitespaceFilterOutputStream extends FilterOutputStream {
    private boolean lastPrintingWasIdchar = false;

    private boolean lastWasSpace = true;

    public CollapseWhitespaceFilterOutputStream(OutputStream sink) {
        super(sink);
    }

    public void write(int code) throws IOException {
        boolean currentIsSpace = Character.isWhitespace(code);
        if(!currentIsSpace) {
            boolean currentIsIdchar = Character.isJavaIdentifierPart(code);
            if(lastWasSpace && lastPrintingWasIdchar && currentIsIdchar)
                super.write(' ');
            super.write(code);

            lastPrintingWasIdchar = currentIsIdchar;
        }
        lastWasSpace = currentIsSpace;
    }
}
```

Listing 1.26: Ausgabefilter, der in Java-Quelltext unnötigen Zwischenraum herausfiltert.

Schließen aller Streams einer Filterkette

Das im Zusammenhang mit `StompJavaSource` (Listing 1.22) aufgetretene Problem ist jetzt gelöst. Eine Anwendung muss nicht mehr jedes Element einer Filterkette einzeln als Ressource definieren und damit einzeln schließen. Es reicht aus, `close` mit dem Kopf einer Filterkette aufzurufen. Durch fortgesetzte Delegation werden dadurch automatisch alle Filter der Kette geschlossen, bis zum konkreten I/O-Objekt am Ende der Kette.

1.4.4 Filter der Laufzeitbibliothek

I/O-Filter in der Bibliothek

Die Laufzeitbibliothek stellt eine Anzahl von Filterströmen zur Verfügung, die häufig anfallende Aufgaben erledigen. Die folgende Liste zeigt eine Auswahl:

`BufferedOutputStream`, `BufferedInputStream`
: Pufferung.

`CheckedOutputStream`, `CheckedInputStream`
: Absicherung durch Prüfsummen.[51]

[51] Siehe Seite 265 für ein Anwendungsbeispiel.

`GZIPOutputStream`, `GZIPInputStream`
: Kompression und Expansion.

`CipherOutputStream`, `CipherInputStream`
: Ver- und Entschlüsseln.

`PrintStream`
: Ausgabe in Textform.

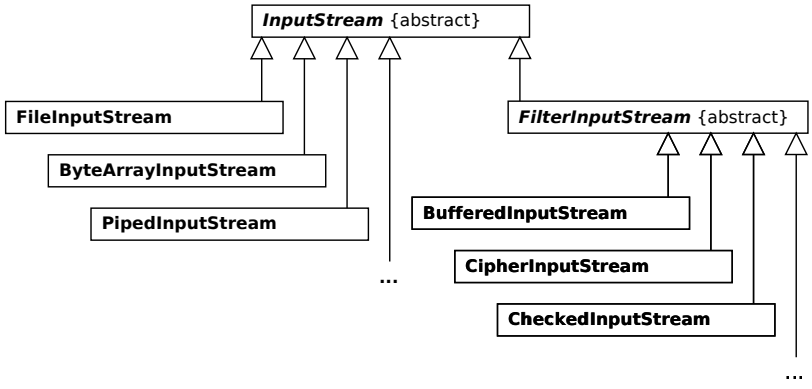
`DataOutputStream`, `DataInputStream`
: Portable Binärdarstellung primitiver Werte.

`ObjectOutputStream`, `ObjectInputStream`
: Serialisieren und Deserialisieren von Objektgraphen (Gegenstand von Kapitel 2).

`LineNumberInputStream`
: Eingabe mit Zeilennummern.

`PushbackInputStream`
: Eingabe mit Wiederholungsmöglichkeit.

Die Filterklassen bringen zum Teil ihre eigene Komplexität mit sich, wie zum Beispiel die `CipherStreams` (siehe Seite 485). Hier sollen deshalb nur einige der einfacheren Filter vorgestellt werden. Die folgende Skizze zeigt die Beziehungen zwischen den `InputStreams`. Die `OutputStreams` sind entsprechend angeordnet.



`BufferedStream`**S**

Konkrete I/O-Klassen, wie zum Beispiel die `FileStreams`, implementieren innerhalb der JVM keine Pufferung. Auf der Ebene des Betriebssystems und darunter greifen natürlich trotzdem Puffer auf verschiedenen Stufen, allerdings ist deren Art

Automatische akzeptable Pufferung

und Umfang unbestimmt. Wie auf Seite 45 beschrieben wurde, lässt sich eine sehr wirksame Pufferung selbst implementieren.

Einen akzeptablen Mittelweg zwischen ungepuffertem und blockweisem I/O bieten die Filterklassen `BufferedInputStream` und `BufferedOutputStream`:

```
import java.io.*;
import static java.lang.System.*;

public class BufferedFileCopy {
    public static void main(String... args) throws IOException {
        try(InputStream input = new BufferedInputStream(new FileInputStream(args[0]));
            OutputStream output = new BufferedOutputStream(new FileOutputStream(args[1]))
            for(int code = input.read(); code >= 0; code = input.read()) {
                output.write(code);
        }
    }
}
```

Listing 1.27: Laufzeitmessung beim Kopieren von Files mit gepufferten Filterströmen.

Der Datendurchsatz reicht nicht an den Durchsatz bei blockweiser Ein- und Ausgabe heran, weil trotz Pufferung für jedes einzelne Byte ein `read`- und ein `write`-Aufruf nötig ist. Dafür muss sich die Anwendung nicht mit der expliziten Verwaltung des Puffer-Arrays auseinandersetzen und arbeitet dennoch mit annehmbarer Geschwindigkeit. Weiter entfällt die nicht portable Festlegung einer bestimmten Puffergröße, weil die `BufferedStream`-Konstruktoren automatisch auf jedem System eine dort brauchbare Puffergröße wählen.

`DataStream`**`s`**

Schreiben und Lesen primitiver Werte

Die I/O-Streams arbeiten ausnahmslos mit Bytes. Will man einen Wert eines anderen primitiven Typs, wie zum Beispiel `int` oder `double`, ausgeben und einlesen, so müsste man den betreffenden Wert in eine Bytefolge zerlegen beziehungsweise wieder daraus rekonstruieren. Diese Aufgabe ist mühsam und schwierig. Die Filter `DataInputStream` und `DataOutputStream` definieren den üblichen Satz von Methoden, darüber hinaus jedoch noch für jeden primitiven Typ eine Methode zum Lesen beziehungsweise Schreiben.

Typ	DataOutputStream	DataInputStream
boolean	void writeBoolean(boolean)	boolean readBoolean()
byte	void write(int)	byte readByte()
char	void writeChar(int)	char readChar()
short	void writeShort(int)	short readShort()
		int readUnsignedShort()
int	void writeInt(int)	int readInt()
long	void writeLong(long)	long readLong()
float	void writeFloat(float)	float readFloat()
double	void writeDouble(double)	double readDouble()
String	void writeUTF(String)	String readUTF()

Die `write`- und `read`-Methoden für jeden Typs arbeiten paarweise zusammen und benutzen die gleiche systemneutrale Bytedarstellung.

Schreiben und Lesen von Strings

Die beiden Methoden in der letzten Zeile fallen etwas aus dem Rahmen, weil Strings keine primitiven Typen sind. Sie werden der Vollständigkeit wegen angeboten. `writeUTF` und `readUTF` erzeugen und lesen die UTF-Darstellung der Einzelzeichen des Strings. Vor die codierten Zeichen selbst schreibt `writeUTF` zusätzlich die Länge der nachfolgenden Bytedarstellung. So ist gewährleistet, dass `readUTF` den String wieder korrekt einlesen kann.

Der zugrunde liegende Bytestrom hat keine Informationen über die Typen der Werte, die darauf geschrieben wurden. Es ist Sache der Anwendung, beim Einlesen wieder passende `read`-Methoden aufzurufen.

Das folgende Programm entscheidet anhand der Kommandozeilenargumente, ob es Daten liest oder schreibt. Beim Aufruf mit einem oder mehreren Argumenten gibt es deren Anzahl als `int`-Wert, dann die einzelnen Argumente nacheinander als Strings auf einen `DataOutputStream` aus. Beim Aufruf ohne Argumente liest es die entsprechenden Daten von einem `DataInputStream` und protokolliert sie auf der Standardausgabe:

```
import java.io.*;

public class DataIO {
    public static void main(final String... args) throws IOException {
        if(args.length > 0)
            // Ausgeben
            try(OutputStream output = System.out;
                DataOutputStream dataOutput = new DataOutputStream(output)) {
                dataOutput.writeInt(args.length);
```

```
                for(String arg: args)
                    dataOutput.writeUTF(arg);
            }
        else
            // Einlesen
            try(InputStream input = System.in;
                DataInputStream dataInput = new DataInputStream(input)) {
                int num = dataInput.readInt();
                while(num-- > 0)
                    System.out.println(dataInput.readUTF());
            }
    }
}
```

Listing 1.28: Schreibt die Kommandozeilenargumente als Java-Objekte in eine Datei oder liest Java-Objekte aus einer Datei.

Das folgende Beispiel zeigt Aufrufe des Programms:

```
$ java DataIO hello world > data
$ java DataIO < data
hello
world
```

GZIPStream**s**

Kompression und Expansion beim Schreiben und Lesen

Die beiden Klassen `GZIPInputStream` und `GZIPOutputStream` sind im Package `java.util.zip` definiert.[52] Sie komprimieren beziehungsweise expandieren einen Bytestrom. Dabei wird das offen dokumentierte und frei verfügbare „GNU Zip“-Kompressionsverfahren angewendet, das viele Werkzeuge erzeugen und verarbeiten können. Dieses Kompressionsverfahren darf nicht mit dem Zip-Archivformat verwechselt werden, das im nächsten Abschnitt diskutiert wird. Die GZip-Kompression erzeugt einen einzigen anonymen Datenstrom und kennt keine Abschnitte, Files, Pfade oder andere Metadaten.

Der folgende Filter erwartet eines der Kommandozeilenargumente `+` oder `-`. Im ersten Fall komprimiert er die Standardeingabe, im zweiten Fall expandiert er sie.

```
import java.io.*;
import static java.lang.System.*;
import java.util.zip.*;

public class CompressedFileCopy {
    public static void main(String... args) throws IOException {
        if(args[0].equals("+"))
            try(InputStream input = in;
```

[52] Der Grund wird im nächsten Abschnitt erklärt.

```
                OutputStream output = new GZIPOutputStream(out)) {
                copyStream(in, out);
            }
        else if(args[0].equals("-"))
            try(InputStream input = new GZIPInputStream(in);
                OutputStream output = out) {
                copyStream(in, out);
            }
        else
            throw new IllegalArgumentException("expected commandline argument + or -");
    }

    private static void copyStream(InputStream from, OutputStream to) throws IOException
        for(int code = from.read(); code >= 0; code = from.read())
            to.write(code);
    }
}
```

Listing 1.29: Komprimiert oder expandiert einen Bytestrom.

GZip-komprimierte Dateien werden üblicherweise mit der Extension „.gz“ versehen, wie im folgenden Beispiel:

```
$ java CompressedFileCopy + < CompressedFileCopy.java > CompressedFileCopy.java.gz
$ java CompressedFileCopy - < CompressedFileCopy.java.gz > CompressedFileCopy.java
```

Im zweiten Schritt wird eine Kopie der Ausgabedatei reproduziert. Ein Blick in den Quelltext zeigt, dass er die Kompression und Expansion unbeschädigt überstanden hat.

Wirksamkeit der Kompression

Ein Vergleich der Filegrößen zeigt, dass die komprimierte Datei kleiner als das Original ist. Der Unterschied nimmt sich bei einem so kurzen Original nicht sehr beeindruckend aus. Die Kompressionsrate hängt einerseits von der Länge des Originals ab, andererseits auch von dessen Inhalt. Wie jedes Kompressionsverfahren nutzt auch die GZip-Kompression Regelmäßigkeit aus. Wenn es keine Regelmäßigkeiten gibt, bleibt die Kompression wirkungslos. Im folgenden Beispiel wird mithilfe von `Write1MegOddBytes` (Listing 1.20) eine Million ungerader Bytes in eine Datei geschrieben und diese dann komprimiert. Von den 8 Bits jedes Bytes ist das niederwertige immer 1, die übrigen sieben weisen keine Regelmäßigkeit auf. Kompression kann die Datei daher um etwa ein Achtel verkleinern:[53]

```
$ java Write1MegOddBytes 1m
$ java CompressedFileCopy + < 1m > 1m.gz
$ ls -1s 1m*
```

[53] Die genaue Kompressionsrate hängt von der konkreten Eingabe ab und schwankt etwas. In diesem Beispiel kommt das Programm bis auf 0,56% an das Optimum von 917504 Bytes heran. Diese Überlegung geht von der Annahme aus, dass die Eingabe tatsächlich aus zufälligen ungeraden Bytes besteht.

```
1048576 1m
 922601 1m.gz
```

Einmal komprimierte Files können nicht noch einmal verkleinert werden, wie das folgende Beispiel zeigt:[54]

```
$ java CompressedFileCopy + < 1m.gz > 1m.gz.gz
$ java CompressedFileCopy + < 1m.gz.gz > 1m.gz.gz.gz
$ ls -1s 1m*
1048576 1m
 922601 1m.gz
 922425 1m.gz.gz
 922700 1m.gz.gz.gz
```

Stattdessen wächst die Filegröße sogar wieder etwas an, weil das GZip-Format ein wenig Verwaltungsinformation erfordert, die der Kompression entgegenwirkt.

Mediendaten im Internet schlecht komprimierbar

Textdateien lassen sich in Regel auf etwa 40% der Originalgröße komprimieren. Die typischen, im Internet verwendeten Medienformate (JPEG-Bilder, MP3-Files, H264-Videos) sind dagegen meist bereits komprimiert. Ein erneuter Kompressionsversuch zeigt keine nennenswerte Wirkung.

1.4.5 Zip-Dateien

Dateiformate komprimierter Daten

Es gibt verschiedene Kompressionsalgorithmen und -formate für unterschiedliche Zwecke und mit unterschiedlichen Eigenschaften. Das GZip-Verfahren, das im vorhergehenden Abschnitt vorgestellt wurde, komprimiert einen einzelnen Datenstrom ohne weitere Zusatzinformationen. Das Zip-Dateiformat komprimiert dagegen mehrere Dateien samt Namen, Pfaden, Kommentaren und ein paar weiteren Metadaten in eine einzige Archivdatei.[55] Es eignet sich daher zum Packen ganzer Directorybäume. Von Zip abgeleitet ist das Java-spezifische Jar-Dateiformat, das weitere Metadaten enthält.[56] Die Laufzeitbibliothek stellt Klassen zum Umgang mit diesen drei Kompressionsformaten bereit[57], die wegen ihrer Komplexität in ei-

[54] Das gilt zumindest beim Einsatz des gleichen Kompressionsverfahrens. Ein besseres Verfahren mag Regelmäßigkeiten aufdecken und ausnutzen, die einem schlechteren Verfahren entgangen sind, und damit eine weitere Verkleinerung erreichen. Allerdings gibt es ein theoretisches Minimum des Restvolumens, das nicht unterschritten werden kann. Anschaulich ausgedrückt: Perfektes „Rauschen“ kann von keinem Kompressionsverfahren mehr verkleinert werden.

[55] Das Zip-Dateiformat reicht nicht für alle Unix-Metadaten aus und ist daher dort nicht so populär wie auf Windows-Systemen. Auf Unix ist traditionell jedes Werkzeug nur für eine einzige Aufgabe zuständig. Entsprechend gibt es unabhängige Werkzeuge für das Packen eines Directorybaums (`tar`) und zur Kompression (`gzip`, `bz2`, `xz` und andere).

[56] Neben Jar gibt es noch das ebenfalls Java-spezifische Dateiformat „Pack200“, das speziell auf die Kompression von Java-Bytecode ausgerichtet ist.

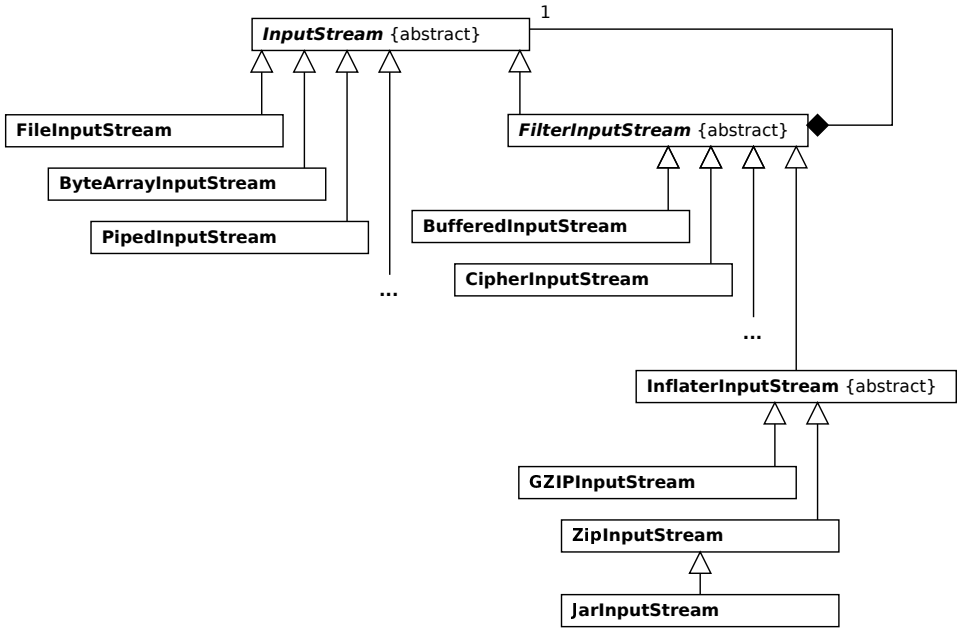
[57] Für viele weitere Kompressionsformate gibt es frei verfügbare Java-Implementierungen.

genen Packages organisiert sind:

GZip	`java.util.zip`
Zip	`java.util.zip`
Jar	`java.util.jar`

Basisklassen für Filter zur Kompression und Expansion

Allen diesen Klassen ist gemeinsam, dass sie Daten mit dem gleichen Algorithmus komprimieren oder expandieren.[58] Deshalb sind sie von den gemeinsamen Basisklassen `InflaterInputStream` und `DeflaterOutputStream` abgeleitet, die wiederum `FilterInputStream`s beziehungsweise `FilterOutputStream`s sind.[59] Die folgende Skizze zeigt den Ausschnitt der `InflaterInputStream`s aus dieser Typenhierarchie:



Zip-Dateien als Sammlungen von Einträgen

In diesem Abschnitt wird der Umgang mit den weitverbreiteten Zip-Dateien vorgestellt.

Ein Zip-Archiv wird über die beiden Klassen `ZipOutputStream` und `ZipInputStream` erzeugt beziehungsweise ausgepackt, ähnlich wie ein GZip-Stream. Anders als ein

[58] Aus diesem Grund sind die Kompressionsraten und Laufzeiten etwa gleich, unabhängig vom Format.

[59] Die ebenfalls definierten Basisklassen `DeflaterInputStream` und `InflaterOutputStream` komprimieren beim Lesen und expandieren beim Schreiben. In der Laufzeitbibliothek gibt es keine konkreten Implementierungen dieser beiden Basisklassen.

GZip-Stream sind Zip-Archive aber in „Einträge“ unterteilt. Einträge eines Zip-Archivs werden durch die Klasse `ZipEntry` repräsentiert. Beim Erzeugen eines Zip-Archivs wird also eine Folge von `ZipEntry`-Objekten und jeweils anschließenden Daten ausgegeben. Entsprechend wird beim Auspacken eine Folge von `ZipEntry`-Objekten mit jeweils nachfolgenden Daten gelesen.

Erzeugen einer Zip-Datei

Jeder Eintrag hat einen Namen, der innerhalb des Archivs eindeutig sein muss. Dieser Name spielt die Rolle eines Pfadnamens, gilt aber nur im Archiv und muss keinen Bezug zur Außenwelt haben. Darüber hinaus können jedem Eintrag weitere Metadaten zugewiesen werden, wie zum Beispiel eine Zeitmarke oder ein Kommentartext. Das folgende Programm liest eine Reihe von Filenamen von der Kommandozeile, packt diese Dateien in ein Zip-Archiv und schreibt es auf die Standardausgabe:

```
import java.io.*;
import static java.lang.System.*;
import java.util.zip.*;

public class CreateZip {
    public static void main(String... args) throws IOException {
        try(ZipOutputStream zipOutput = new ZipOutputStream(out)) {
            for(String arg: args)
                try(InputStream input = new FileInputStream(arg)) {
                    ZipEntry entry = new ZipEntry(arg);
                    zipOutput.putNextEntry(entry);
                    for(int code = input.read(); code >= 0; code = input.read())
                        zipOutput.write(code);
                }
        }
    }
}
```

Listing 1.30: Verpackt die auf der Kommandozeile angegebenen Dateien in ein Zipfile.

Auspacken einer Zip-Datei

Beim Lesen eines Zip-Archivs wird mit der Methode `getNextEntry` der jeweils nächste Zip-Eintrag angesteuert, bis die Methode `null` liefert und damit anzeigt, dass keine weiteren Einträge mehr folgen. Der Inhalt jedes Eintrags wird mit `read`-Methoden gelesen. Am Ende jedes Eintrags wird das Ergebnis -1 zurückgegeben, obwohl nicht der ganze `ZipInputStream`, sondern nur der betreffende Eintrag erschöpft ist.

Das Programm `ExplodeZip` liest ein Zip-Archiv von der Standardeingabe und kopiert alle darin gefundenen Einträge in Files mit dem entsprechenden Pfadnamen:

```
import java.io.*;
```

```
import java.util.zip.*;

public class ExplodeZip {
    public static void main(String... args) throws IOException {
        try(ZipInputStream zis = new ZipInputStream(System.in)) {
            for(ZipEntry ze = zis.getNextEntry(); ze != null; ze = zis.getNextEntry())
                try(OutputStream output = new FileOutputStream(ze.getName())) {
                    for(int code = zis.read(); code >= 0; code = zis.read())
                        output.write(code);
                }
        }
    }
}
```

Listing 1.31: Packt ein Zipfile aus.

Zip-Einträge enthalten einige Metadaten, die dem `ZipEntry`-Objekt entommen und zum Teil auch dort eingetragen werden können

Metadaten in Zip-Dateien

`String getName()`
: Name.

`int getMethod()`
: Kompressionsalgorithmus. Zip-Archive können auch unkomprimierte Daten speichern. In diesem Fall liefert diese Methode den Wert `ZipEntry.STORED`, im Gegensatz zu `ZipEntry.DEFLATED` bei komprimierten Archiven.

`String getComment()`
: Kommentar zu diesem Eintrag. Kommentare können nicht ohne Weiteres aus einem `ZipInputStream` gelesen werden, weil sie gemäß Datenformat gesammelt am Ende des Archivs abgelegt sind. Diese Eigenschaft gilt unabhängig von einer bestimmten Implementierung. In einem beliebigen `ZipInputStream` steht die Information noch nicht zur Verfügung, wenn ein Eintrag geöffnet wird. Daher liefert die Methode `getComment` den Wert `null`.[60]

`long getCompressedSize()`
: Platzbedarf im komprimierter Form, das heißt im Archiv.

`long getSize()`
: Platzbedarf in unkomprimierter Form.

`byte[] getExtra()`
: Zusätzliche Informationen. Die Extra-Informationen können nach Bedarf verwendet werden und haben keine feste Bedeutung.

[60] Die ebenfalls im Package `java.util.zip` definierte Klasse `ZipFile` repräsentiert keinen beliebigen Datenstrom mit einem Zip-Archiv, sondern eine Datei, die komplett eingelesen wird. Dort stehen die Kommentare zur Verfügung.

`long getTime()`
: Zeitmarke.

`boolean isDirectory()`
: Eintrag entspricht einem Directory.

1.5 Decorator-Pattern

Zusammenspiel der I/O-Klassen

Die Trennung von konkreten I/O-Klassen (`FileStream`, `ByteArrayStream`, `PipedStream`) und Filterklassen (`BufferedStream`, `DataStream`, `GZIPStream`) erweist sich als flexible Konstruktion. Zur Laufzeit können beliebige Filterobjekte an ein konkretes I/O-Objekte gekoppelt werden. Damit kann eine komplexe Verarbeitungskette aus einfachen Bausteinen zusammengesetzt werden.

Wiederkehrende Struktur und Bausteine

Diese Idee lässt sich auch in anderen Situationen anwenden. Die Konstruktion weist dabei immer die gleiche Struktur auf und wird deshalb als „Entwurfsmuster“ oder „Schablone“ (*pattern*) bezeichnet. In diesem Fall handelt es sich um das sogenannte **Decorator-Pattern**. Das Decorator-Pattern hat die folgenden Merkmale:

- Alle Bausteine weisen die gleiche Funktionalität auf. Diese lässt sich mit einem Interface oder einer abstrakten Basisklasse festlegen.
- Es gibt eine Reihe von elementaren, konkreten Bausteinen. Diese sind isoliert lebensfähig und können einzeln eingesetzt werden. Sie sind als konkrete Klassen definiert, die das Interface implementieren beziehungsweise die abstrakte Basisklasse ableiten.
- Es gibt weiter eine Sammlung von Bausteinen, die die Grundfunktionalität in irgendeiner Weise erweitern oder verändern. Sie kapseln („dekorieren“) einen anderen Baustein der gleichen Art und werden deshalb als „Dekoratoren“ bezeichnet. Dekoratoren können ihre Leistung nicht alleine erbringen, sondern hängen immer von einem anderen Baustein ab, dessen Funktionalität sie einerseits nutzen und andererseits verändert wieder zur Verfügung stellen.
- Allen Dekoratoren ist die Kapselung eines dekorierten Bausteins gemeinsam. Diese Gemeinsamkeit kann in einer abstrakten Dekorator-Basisklasse zusammengefasst werden, die die eigentlichen Dekoratorklassen vereinfacht.

Allgemeine Bezeichnungen der Komponenten

Im Zusammenhang mit dem Decorator-Pattern haben sich die folgenden Bezeichnungen etabliert:

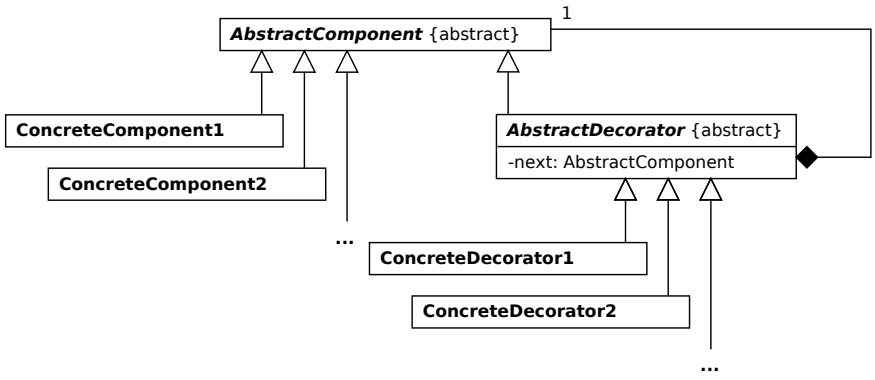
AbstractComponent
: Gemeinsames Interface oder gemeinsame ABC.

ConcreteComponent
: Elementare, konkrete Bausteine.

AbstractDecorator
: ABC aller Dekoratoren, verwaltet den dekorierten Baustein.

ConcreteDecorator
: Bestimmte Dekoratoren, die die Funktionalität auf ihre Weise verändern.

Das folgende UML-Klassendiagramm zeigt die Beziehungen zwischen diesen Typen:



Rollen der I/O-Klassen aus Sicht des Musters

Im Bezug auf Byteströme zur Eingabe entsprechen den allgemeinen Bezeichnungen des Decorator-Patterns die folgenden Java-Typen:

AbstractComponent
: `InputStream`

ConcreteComponent
: `FileInputStream`, `ByteArrayInputStream`, ...

AbstractDecorator
: `FilterInputStream`

ConcreteDecorator
: `BufferedInputStream`, `DataInputStream`, `GZIPInputStream`, ...

Symmetrische Strukturen auf Ein- und Ausgabeseite

Eine zweite, weitgehend gleichartige Struktur findet sich auf der Ausgabeseite. Auch diese Klassen sind nach dem Decorator-Pattern organisiert. Das Gleiche gilt für die Klassen zur Textein- und -ausgabe, die im nächsten Abschnitt vorgestellt werden.

Ein komplett anderes Beispiel des Decorator-Patterns finden Sie in Anhang C.

1.6 Umgang mit Textdateien

1.6.1 Interne und externe Darstellung

Textdarstellung primitiver Werte

Im Quelltext, auf der Kommandzeile und bei anderen Ein- und Ausgaben wird eine lesbare Schreibweise von primitiven Werten benutzt. Ganze Zahlen werden zum Beispiel als Folge von Dezimalziffern geschrieben und bei Bedarf mit einem Vorzeichen versehen. Etwas komplizierter sind Floatingpoint-Literale aufgebaut. Aber auch hier wird die gleiche Textschreibweise, die der Compiler im Quelltext akzeptiert, auch zur Laufzeit gelesen und ausgegeben, wie das folgenden Beispielprogramm zeigt:

```
public class CmdlineArgs {
    public static void main(String... args) {
        int arg = 0;
        int intValue = Integer.parseInt(args[arg++]);
        double doubleValue = Double.parseDouble(args[arg++]);
        boolean booleanValue = Boolean.parseBoolean(args[arg++]);
        System.out.println(intValue);
        System.out.println(doubleValue);
        System.out.println(booleanValue);
    }
}
```

Listing 1.32: Lesen der Textdarstellung (Unparsing) primitiver Werte von der Kommandozeile.

Einlesen der Textdarstellung von der Kommandozeile

Der folgende Aufruf reproduziert die eingegebenen Werte:[61]

```
$ java CmdlineArgs +23 -3.14e-6 true
23
-3.14E-6
true
```

Binärdarstellung als Alternative

Diese lesbare Darstellung von Daten wird als externe oder **Textdarstellung** bezeichnet. Sie eignet sich einerseits zur Kommunikation zwischen Programmen und Menschen, andererseits auch zur Kommunikation zwischen Programmen auf verschiedenen Systemen oder in verschiedenen Implementierungssprachen. Die externe Darstellung ist dagegen schlecht geeignet für Rechenoperationen im laufenden Programm.

Innerhalb eines Programms wird eine interne oder auch **binäre Darstellung** benutzt, die als Bitmuster in Bytes abgelegt ist. Die interne Darstellung lässt sich effizient verarbeiten, ist aber weder für menschliche Leser noch für beliebige andere

[61] Das kleine `e` und das große `E` zur Markierung des Zehnerexponenten sind gleichwertig.

Programme gedacht. Allenfalls lässt sich die interne Darstellung zum Datenaustausch zwischen zwei Java-Programmen nutzen.[62] Zum Beispiel schreiben und lesen die `DataStream`-Filterklassen (siehe Seite 64) die binäre Darstellung primitiver Werte.

Parsing und Unparsing

Die Umwandlung von der externen in die interne Darstellung wird als **Parsing** bezeichnet, die gegenläufige Umwandlung der internen in die externe Form als **Unparsing**. Beim Übersetzen von Java-Quelltext „parst" der Compiler zuerst die externe Darstellung des ganzen Programms, einschließlich der literalen Textdarstellung primitiver Werte, in eine interne Darstellung, die dann der weiteren Transformation in Bytecode zugeführt wird.

Beispiel: `double`-Wert *pi*

Am Beispiel der Kreiszahl *pi* soll der Unterschied zwischen der internen und der externen Darstellung deutlich gemacht werden. Die Kreiszahl ist eine transzendente Zahl. In Java wird sie durch einen `double`-Wert approximiert, der die genauest mögliche Annäherung an den mathematischen Wert darstellt. Dieser `double`-Wert hat die Textdarstellung

```
3.141592653589793
```

und kann in dieser Form beispielsweise im Dialog mit der Tastatur eingetippt, auf der Kommandozeile angegeben, auf dem Bildschirm ausgebeben oder auf Papier gedruckt werden. Die interne Darstellung des `double`-Werts besteht aus 64 Bits mit der hexadezimalen Schreibweise[63]

```
400921FB54442D18
```

Zur Vereinfachung ist dieser Wert in der Variablen `Math.PI` gespeichert.

Merkmale Text- und Binärdarstellung

Interne und externe Darstellung haben spezifische Vor- und Nachteile, aus denen sich die Anwendungszwecke ergeben:

- Interne Darstellung im laufenden Programm: effiziente Verarbeitung, kompakt, eindeutig.
- Externe Darstellung bei der Ein- und Ausgabe: portabel, lesbar, systemneutral.[64]

[62] Bei Java spielt es keine Rolle, ob die beiden Programme auf dem gleichen oder auf verschiedenen Rechnern laufen. Bei andere Programmiersprachen gilt das nicht immer. Zum Beispiel hängt die interne Darstellung einiger primitiver Typen in C von der Plattform ab.

[63] Die Interpretation dieser Bits ist als „IEEE 754 floating-point double format" dokumentiert.

[64] Darüber hinaus kann die Textdarstellung ausgedruckt oder abgeschrieben werden. Das eröffnet die Möglichkeit der langfristigen Archivierung. Bisher hat kein anderes Speichermedium eine vergleichbare Lebensdauer erreicht wie ausgedruckte Listings.

Byteströme vs. Textdaten

Die bisher in diesem Kapitel vorgestellten Klassen beruhen alle auf den ABCs `InputStream` und `OutputStream` (Seite 28). Diese sind für Byteströme ausgelegt und eignen sich damit vorrangig zur Übermittlung von binären Daten. Neben der Verarbeitung von binären Daten ist aber die Verarbeitung von Textdaten genauso wichtig. Deshalb gibt es in Java zusätzliche I/O-Klassen, die speziell zur Verarbeitung von Texten gedacht sind.

1.6.2 Basisklassen `Reader` und `Writer`

ABCs zur Texteingabe und -ausgabe

Die beiden abstrakten Basisklassen `Reader` und `Writer` (beide im Package `java.io` definiert) legen die Methoden zur Ein- und Ausgabe von Textdaten fest. Die beiden Klassen sind *nicht* von `InputStream` und `OutputStream` abgeleitet. Sie sind keine Byteströme, sondern Textströme, und transportieren keine Bytes, sondern Textzeichen. Die Klassen definieren die folgenden elementaren Methoden:

Methoden von `Reader` und `Writer`

- `Reader`

  `int read()`
  : Liefert das nächste Zeichen oder -1, wenn die Eingabe erschöpft ist.

  `int read(char[] buffer)`
  : Füllt den Puffer soweit möglich und liefert die Anzahl tatsächlich gelesener Zeichen oder -1, wenn nichts mehr gelesen werden konnte und die Eingabe erschöpft ist.

  `int read(char[] buffer, int start, int num)`
  : Wie `read(char[])`, füllt aber nur den Ausschnitt des Puffers beginnend mit dem Index `start` mit der Länge `num`.

  `void close()`
  : Schließt die Eingabe.

- `Writer`

  `void write(int chr)`
  : Schreibt das Zeichen mit dem Code `chr` auf die Ausgabe.

  `void write(char[] buffer)`
  : Gibt alle Zeichen des Puffers aus.

  `void write(char[] buffer, int start, int num)`
  : Gibt den Ausschnitt des Puffers beginnend mit dem Index `start` mit der Länge `num` aus.

`void write(String text)`
: Gibt alle Zeichen des Strings aus.

`void write(String text, int start, int num)`
: Gibt einen Ausschnitt des Strings aus.

`void flush()`
: Entleert alle Puffer.

`void close()`
: Schließt die Ausgabe.

Transport von Textzeichen statt Bytes

Die Methoden funktionieren genauso wie die Gegenstücke bei den Byteströmen. Im Gegensatz zu jenen transportieren sie aber Textzeichen und keine Bytes. Der Ergebniswert von `read()` liegt im Bereich 0 bis 65535 (= 2^{16}-1), zuzüglich des Fluchtwerts -1 zur Anzeige des Eingabeendes. Symmetrisch dazu schreibt `write(int chr)` das Zeichen mit dem Unicode `chr` auf die Ausgabe, wobei nur die niederwertigen 16 Bits des Arguments verwendet und die höherwertigen Bits ignoriert werden. Die `write`-Methode ist für Zeichen-Arrays und Strings überladen.

`flush` und `close` spielen die gleiche entscheidende Rolle wie bei Byteströmen. Fehlende Aufrufe können leicht zu unvollständigen Ausgaben und schwer behebbaren Laufzeitfehlern führen. `Reader` und `Writer` implementieren das Interface `Closeable` und sollten mit ARM eingesetzt werden.

1.6.3 Konkrete Quellen und Senken

Konkrete Klassen zur Textein- und -ausgabe

Von den ABCs `Reader` und `Writer` abgeleitet sind verschiedene Klassen mit konkreten Textausgabezielen beziehungsweise -eingabequellen:

`FileReader`, `FileWriter`
: File im Filesystem.

`CharArrayReader`, `CharArrayWriter`
: `char`-Array im eigenen Programm.

`StringReader`, `StringWriter`
: String im eigenen Programm.

`PipedReader`, `PipedWriter`
: Anderer Thread (siehe Kapitel 6, Seite 361).

Wie die Klassen `FileInputStream` und `FileOutputStream` erwarten `FileReader` und `FileWriter` Pfadnamen als Argumente. `FileWriter` bietet außerdem überladene Konstruktoren zum wahlweisen Verlängern einer vorhandenen Datei.

Ein- und Ausgabe mit Strings und `char`-Arrays

Die Klassen `StringReader` und `StringWriter` sowie `CharArrayReader` und `CharArrayWriter` lesen und schreiben Zeichenfolge aus den beziehungsweise in die entsprechenden Datenstrukturen. Die Writer definieren Default-Konstruktoren und erzeugen und verwalten den nötigen internen Speicher dann selbst. Der Benutzer ist von Kapazitätsabschätzungen entbunden. Diese Eigenschaft macht die Klassen attraktiv für Zwischenspeicher. Die Reader erwarten einen String beziehungsweise ein Zeichen-Array und machen dessen Inhalt als Reader zur Verfügung.

Ändern einer Datei *in place*

Das folgende Beispielprogramm entfernt alle Blockkommentare aus einer Java-Quelltextdatei. Der Inhalt der Datei soll an Ort und Stelle ersetzt werden. Das Öffnen *ein und derselben* Datei zum gleichzeitigen Lesen und Schreiben löscht den Inhalt der Datei. Deshalb wird der Inhalt der Datei zunächst gelesen, verarbeitet und das Ergebnis in einem lokalen `StringWriter` zwischengespeichert. Erst nachdem die komplette Eingabe verarbeitet ist, wird der Inhalt von `StringWriter` mit einem `StringReader` wieder gelesen und auf die ursprüngliche Datei zurückgeschrieben. Der konkrete Algorithmus, der sich im Wesentlichen in der ersten `for`-Schleife findet, steht hier nicht im Mittelpunkt.[65]

```
import java.io.*;

public class Xcomment {
    private static enum State {
        Out, Into, In, Outof
    };

    public static void main(String... args) throws IOException {
        State state = State.Out;
        try(Reader reader = new FileReader(args[0]);
            StringWriter writer = new StringWriter()) {
            for(int chr = reader.read(); chr >= 0; chr = reader.read())
                switch(state) {
                    case Out:
                        if(chr == '/')
                            state = State.Into;
                        else
                            writer.write(chr);
                        break;
                    case Into:
                        if(chr == '*')
                            state = State.In;
                        else if(chr == '/')
                            writer.write('/');
                        else {
                            writer.write('/');
                            writer.write(chr);
                            state = State.Out;
                        }
                        break;
```

[65] Der Algorithmus ist simpel gebaut und kommt beispielsweise nicht mit Kommentaren in String-Literalen zurecht.

```
                    case In:
                        if(chr == '*')
                            state = State.Outof;
                        break;
                    case Outof:
                        if(chr == '/') {
                            writer.write(' ');
                            state = State.Out;
                        }
                        else if(chr != '*')
                            state = State.In;
                        break;
                }
            try(Reader rereader = new StringReader(writer.toString());
                Writer rewriter = new FileWriter(args[0])) {
                for(int chr = rereader.read(); chr >= 0; chr = rereader.read())
                    rewriter.write(chr);
            }
        }
    }
}
```

Listing 1.33: Löschen von Blockkommentaren aus einer Quelltextdatei *in place*.

Dieses Vorgehen demonstriert den Einsatz der Klassen `StringWriter`, `StringReader` und `FileWriter`.

1.6.4 Textfilter

Filterklassen für die Textein- und -ausgabe

Die Klassenhierarchien und Reader und Writer sind nach dem Decorator-Pattern organisiert, wie die Byteströme. Die beiden Klassen `FilterReader` und `FilterWriter` übernehmen die Rolle der abstrakten Dekoratoren und dienen als Ausgangspunkt zur Implementierung von Textfiltern.

Textfilter in der Bibliothek

Die Laufzeitbibliothek stellt bereits einige Textfilter zur Verfügung:[66]

`BufferedReader`
: Gepuffertes Einlesen auf Zeilenbasis.

`BufferedWriter`
: Gepufferte Ausgabe.

`PrintWriter`
: Formatierte Ausgabe und Ausgabe primitiver Werte.

[66] Diese vordefinierten Textfilter sind nicht von den entsprechenden Filterklassen abgeleitet. Auf die Anwendung hat das keinen Einfluss.

Zeilenweise Eingabe

Zeilenweises Lesen von Text mit `BufferedReader`

Die Klasse `BufferedReader` puffert die Eingabe eines vorgeschalteten Readers auf der Grundlage von Zeilen. Über die Reader-Methoden hinaus definiert `BufferedReader` insbesondere die Methode `readLine`, die bei jedem Aufruf den Inhalt einer weiteren Textzeile zurückliefert. Am Ende der Eingabe ist das Ergebnis `null`. `BufferedReader` erkennt die plattformspezifische Codierung von Zeilenwechseln.[67]

Beispiel: Zeilennummerierung

Das folgende Programm `LineNumbers` kopiert eine Textdatei und setzt an den Anfang jeder Zeile die Zeilennummer:

```
import java.io.*;

public class LineNumbers {
    public static void main(String... args) throws IOException {
        try(BufferedReader reader = new BufferedReader(new FileReader(args[0]));
            Writer writer = new FileWriter(args[1])) {
            int num = 1;
            for(String line = reader.readLine(); line != null; line = reader.readLine())
                writer.write(num + "\t");
                writer.write(line);
                writer.write('\n');
                num++;
            }
        }
    }
}
```

Listing 1.34: Kopieren einer Textdatei mit Einfügen von Zeilennummern.

Der folgende Aufruf versieht den eigenen Quelltext mit Zeilennummern:

```
$ java LineNumbers LineNumbers.java numbered
$ cat numbered
1       import java.io.*;
2
3       public class LineNumbers {
4           public static void main(String... args) throws IOException {
5               try(BufferedReader reader = new BufferedReader(new FileReader(args[0]));
6                   Writer writer = new FileWriter(args[1])) {
7                   int num = 1;
8                   for(String line = reader.readLine(); line != null; line = reader.read
9                       writer.write(num + "\t");
10                      writer.write(line);
11                      writer.write('\n');
12                      num++;
```

[67] Leider verwenden verschiedene Systeme unterschiedliche Kontrollzeichen oder Kombinationen von Kontrollzeichen, um Zeilenwechsel zu codieren. `BufferedReader` kapselt diese Unterschiede und ist damit über Systeme hinweg portabel.

```
13                  }
14              }
15          }
16      }
```

Das Programm geht zwar auf portable Art mit Zeilenwechseln in der *Eingabe* um, schreibt aber auf die *Ausgabe* ein Newline-Zeichen (`'\n'`, Code 10), die Unix-typische Zeilenwechsel-Codierung. Andere Systeme kommen damit oft, aber nicht immer zurecht. Der nächste Abschnitt zeigt eine zuverlässigere Lösung dieses Problems.

Unportable Ausgabe von Zeilenwechseln

Ausgabe der Textdarstellung

Zur Umwandlung der Binär- in die Textdarstellung gibt es verschiedene Wege. Bequem, aber nicht besonders sparsam ist die Verkettung eines Leerstrings mit einem primitiven Wert:[68]

Unparsing von primitiven Werten

```
writer.write("" + lineNumber);
```

Effizienter liefern die statischen `toString`-Methoden der Wrapperklassen die Textdarstellung von primitiven Typen, wie zum Beispiel:

Effiziente Methoden der Wrapperklassen

```
writer.write(Integer.toString(lineNumber));
```

Jede der acht Wrapperklassen stellt eine solche `toString`-Methode zur Verfügung:[69]

```
static String  toString(boolean)      // Boolean
static String  toString(byte)         // Byte
static String  toString(char)         // Character
static String  toString(short)        // Short
static String  toString(int)          // Integer
static String  toString(long)         // Long
static String  toString(float)        // Float
static String  toString(double)       // Double
```

Diese Methoden implementieren das „Unparsing“ primitiver Werte.

Im Zusammenhang mit der Ein- und Ausgabe bietet sich die Filterklasse `PrintWriter`

Filterklasse `PrintWriter` zur Textausgabe primitiver Werte

[68] Insgesamt sind hierzu bis zu drei String-Objekte nötig: Zum einen der Leerstring selbst, dazu der temporäre String, den der Compiler als zweites Argument zur Stringverkettung erzeugt, und schließlich das Ergebnis der Stringverkettung.

[69] Diese statischen `toString`-Methoden haben, abgesehen vom gleichlautenden Namen, nichts mit der normalen (dynamischen) `toString`-Methode zu tun, die in `Object` vordefiniert ist und die von den meisten Klassen redefiniert wird.

an. `PrintWriter` definiert die drei Methoden `print`, `println` und `printf`. Die ersten beiden sind für Argumente der primitiven Typen sowie der Typen `String`, `char`-Array und `Object` überladen. Sie geben jeweils eine Textdarstellung des Arguments auf den zugrunde liegenden `Writer` aus. Ein Zeichen-Array wird elementweise ausgegeben, bei `Object` wird zuerst die Methode `toString` aufgerufen und deren Ergebnis ausgegeben.

```
void  println()
void  println(boolean)
void  println(byte)
void  println(char)
void  println(short)
void  println(int)
void  println(long)
void  println(float)
void  println(double)
void  println(String)
void  println(char[])
void  println(Object)
PrintWriter  printf(String fmt, Object... args)
PrintWriter  format(String fmt, Object... args)
```

Zu allen `println`-Methoden, abgesehen von der parameterlosen, gibt es auch eine entsprechende `print`-Methode, die keinen Zeilenwechsel nachschiebt.

Formatierte Ausgabe via `PrintWriter`

`printf` ruft die Methode `format` der gleichen Klasse auf. Beide erwarten als erstes Argument einen Formatstring, dazu weitere Argumente passend zu den Formatangaben im Formatstring. Auf die Formatierung selbst wird hier nicht näher eingegangen. Anders als `print` und `println` geben `printf` und `format` den `PrintWriter` selbst zurück und erlauben damit Kettenaufrufe.

Portable Ausgabe von Zeilenwechseln

`println` schließt die Ausgabe mit einem Zeilenwechsel ab. Die Codierung des Zeilenwechsels entspricht dabei der jeweils zugrunde liegenden Plattform. Das Gleiche bewirkt die Formatangabe `%n` in einem Formatstring. Diese Art der Zeilenausgabe ist portabler als die Ausgabe von literalen Newline-Zeichen. Eine verbesserte Fassung von `LineNumbers` (Listing 1.34) benutzt zur Ausgabe einen `PrintWriter` und dessen Methode `printf`:

```
import java.io.*;

public class LineNumbersPrintWriter {
    public static void main(String... args) throws IOException {
        try(BufferedReader reader = new BufferedReader(new FileReader(args[0]));
            PrintWriter writer = new PrintWriter(new FileWriter(args[1]))) {
            int number = 1;     // aktuelle Zeilennummer
            for(String line = reader.readLine(); line != null; line = reader.readLine())
                writer.printf("%d\t%s%n", number, line);
                number++;
```

```
            }
        }
    }
}
```

Listing 1.35: Kopieren einer Textdatei mit Einfügen von Zeilennummern und portablem Umgang mit Zeilenwechseln.

1.6.5 Ersetzen der Standardein- und -ausgabe-Objekte

Abgesehen von der Klasse `PrintWriter` gibt es noch die Filterklasse `PrintStream` (siehe Seite 62). Die Variablen `System.out` und `System.err` zur Standardausgabe und -Fehlerausgabe (Seite 17) haben beide den Typ `PrintStream`. `PrintStream` und `PrintWriter` definieren die gleiche Familie der `print`-Methoden und dienen praktisch dem gleichen Zweck. `PrintStream` ist allerdings älter als `PrintWriter` und existiert seit den frühesten Java-Versionen. Aus technischer Sicht ist diese duplizierte Funktionalität nicht mehr nötig, wird aber aus Gründen der Kompatibilität aufrechterhalten.

Die drei Standardein- und -ausgabe-Objekte `System.in`, `System.out` und `System.err` sind in der Voreinstellung mit der Konsole verknüpft. Das Betriebssystem kann diese Zuordnung durch Ein- und Ausgabe-Umlenkung (Abschnitt 1.1.2) ändern, sodass Programme von Textdateien lesen und darauf schreiben, ohne es zu wissen.

Ein Java-Programm kann einen vergleichbaren Effekt auch aus eigener Kraft erzielen. Die drei statischen `System`-Methoden

```
static void  setIn(InputStream in)
static void  setOut(PrintStream out)
static void  setErr(PrintStream err)
```

weisen beliebige neue Objekt an die Variablen `System.in`, `System.out` und `System.err` zu. Ein Programm kann damit die Standardein- und -ausgabe nach Belieben steuern. Das folgende Beispielprogramm ruft das bekannte Einstiegsprogramm `Hello` (nicht abgedruckt) auf, das seinen Gruß auf die Standardausgabe schreibt, fängt aber dessen Ausgabe ein.

```
import java.io.*;
import static java.lang.System.*;

public class CaptureHello {
    public static void main(String[] args) throws IOException {
        try(ByteArrayOutputStream bytesOutput = new ByteArrayOutputStream();
            PrintStream printStream = new PrintStream(bytesOutput)) {
            PrintStream systemOut = out;     // Original retten
            setOut(printStream);
```

```
            Hello.main(args);
            setOut(systemOut);              // Original restaurieren
            out.println("Captured: " + bytesOutput.toString());
        }
    }
}
```

Listing 1.36: Auffangen der Standardausgabe eines anderen Java-Programmes.

Die drei Methoden `setIn`, `setOut` und `setErr` sind bemerkenswert, weil sie den Wert von `final`-Variablen ändern, wie das folgende Programm nachweist:

```
public class PrintSystemOut {
    public static void main(String[] args) {
        System.err.println(System.out);
        System.setOut(System.err);
        System.err.println(System.out);
    }
}
```

Listing 1.37: Ändern der `final`-Variablen `System.out`.

Die Ausgabe zeigt zwei verschiedene Werte der `final`-Variablen `System.out`:

```
$ java PrintSystemOut
java.io.PrintStream@eb607d
java.io.PrintStream@10bbf6d
```

1.6.6 Zeichensätze und Encodings

Zeichensätze und Encodings zur Abbildung von Textzeichen in Bytes

Bei Textdaten, das heißt `char`-Werte und Strings, die innerhalb eines Java-Programms verarbeitet oder nur zwischen Java-Programmen ausgetauscht werden, spielt die Darstellung keine Rolle. Die Daten sind in einer internen Darstellung gespeichert und verbleiben in dieser. Bei der Ein- und Ausgabe müssen diese Textdaten aber in Bits und Bytes abgebildet werden. Die Verknüpfung zwischen Textzeichen, Codes und Bytes regeln Zeichensätze und Encodings.

Zeichensätze US-ASCII, ISO-8859-1, Unicode

Der US-ASCII-Zeichensatz (*American Standard Code for Information Interchange*) umfasst 128 Zeichen mit Codes von 0 bis 127 und ist auf US-amerikanische Bedürfnisse ausgerichtet. Jedem dieser Zeichen kann direkt ein Byte zugeordnet werden. ISO-8859-1 (auch „Latin-1“) erweitert den US-ASCII-Zeichensatz um weitere 128 Zeichen mit Codes 128–255, die sich an westeuropäischen Sprachen orientieren. In einer globalisierten Welt reichen aber beide Zeichensätze nicht aus. Daher arbeitet

Java mit „Unicode“, der Codes für Schriftzeichen der meisten Sprachen der Welt festlegt.[70]

Beispiel: Code des Umlauts „ü“

Der deutsche Umlaut „ü“ hat beispielsweise den Code 252, wie die folgende Ausgabe zeigt:

```
int code = 'ü';
System.out.println(code);        // 252
```

Verbindlich verfügbare Encodings

Wie der `char`-Wert 252 tatsächlich in Bytes umgesetzt wird, regelt der Unicode nicht. Diese Abbildung ist Sache von **Encodings**, von denen verschiedene im Gebrauch sind.[71] Jede Java-Implementierung muss wenigstens die folgenden „Pflicht-Encodings“ unterstützen:[72]

Encoding	Codebereich	Anzahl Bytes pro Zeichen	Bytefolge von „ü“
`US-ASCII`	0–127	1	nicht darstellbar
`ISO-8859-1`	0–255	1	`FC`
`UTF-8`	Unicode	1–4	`C3 BC`
`UTF-16BE`	Unicode	2, 4	`00 FC`
`UTF-16LE`	Unicode	2, 4	`FC 00`
`UTF-16`	Unicode	2, 4	`FE FF 00 FC`

Der Code 252 für das Zeichen ü wird in jedem dieser Encodings in eine andere Bytefolge abgebildet, wie die rechte Spalte der Tabelle zeigt.

Encodings mit konstanter und mit variabler Länge

Die Encodings wenden unterschiedlich viele Bytes pro codiertem Zeichen auf. Während US-ASCII und ISO-8859-1 konstant mit einem Byte pro Zeichen auskommen, nimmt ein Zeichen in den UTF-Encodings eine wechselnde Anzahl von Bytes ein. Das erschwert die Navigation in einem Bytestrom auf der Grundlage von Textpositionen. Andererseits können Texten mit überwiegend oder ausschließlich US-ASCII-Zeichen kompakt dargestellt werden. Die Bytedarstellung eines Textes, der nur US-ASCII-Zeichen enthält, ist in den Encodings US-ASCII, UTF-8 und ISO-8859-1 identisch. Für Texte mit anderen als nur US-ASCII-Zeichen gilt das nicht.

System-spezifisches Default-Encoding

Das „Default-Encoding“ jedes Systems regelt, welches Encoding bei der Ein- und

[70] Der Unicode wurde laufend erweitert und umfasst in Version 4 etwa 100000 Zeichen.

[71] US-ASCII und ISO-8859-1 sind jeweils Zeichensatz und Encoding in einem. Das Encoding ist in diesen Fällen trivial und ordnet jedem Code das gleichwertige Byte zu. Unicode ist dagegen nur ein Zeichensatz und kein Encoding. Entsprechend sind die verschiedenen UTF-Varianten nur Encodings, aber keine Zeichensätze.

[72] Darüber hinaus steht es jeder Java-Implementierung frei, beliebige weitere Encodings anzubieten.

Ausgabe benutzt wird, wenn keine Angaben gemacht werden. In Java repräsentiert die Klasse `Charset` im Package `java.nio.charset` ein Encoding. Die statischen Methoden `defaultCharset` und `availableCharsets` dieser Klasse geben Auskunft über das aktuelle Default-Encoding und den Katalog aller verfügbaren Encodings. Dieser Katalog liegt nicht fest, sondern kann sich von System zu System unterscheiden. Das Programm `ShowCharsets` gibt das Default-Encoding und den Katalog aller verfügbaren Encodings aus:

Katalog aller verfügbaren Encodings

```
import java.nio.charset.*;

public class ShowCharsets {
    public static void main(String... args) {
        System.out.println(Charset.defaultCharset());
        System.out.println(Charset.availableCharsets().keySet());
    }
}
```

Listing 1.38: Auflisten der Encodings.

Es liefert beispielsweise die folgende Ausgabe (gekürzt):

```
$ java ShowCharsets
ISO-8859-1
[Big5, Big5-HKSCS, EUC-JP, EUC-KR, GB18030, ...]
```

Die Liste der tatsächlich verfügbaren Encodings kann ziemlich lang sein. Sie enthält aber auf jeden Fall die sechs oben aufgezählten, verbindlich vorgeschriebenen Pflicht-Encodings US-ASCII, ISO-8859-1, UTF-8, UTF-16BE, UTF-16LE und UTF-16.

▶ Der primitive Typ `char` umfasst vorzeichenlose 16-Bit-Werte von 0 bis 65535 = 2^{16}-1. In den Anfangsjahren von Java reichte das für den gesamten Unicode aus, aber seit der Umstellung auf Unicode 4 in Java-Version 5 ist der Wertebereich von `char` zu klein.

Der größte Teil der Unicodezeichen wird weiterhin durch einzelne `char`-Werte repräsentiert, aber für einige Zeichen werden zwei `char`-Werte gebraucht. Dabei sind Verwechslungen ausgeschlossen, weil bei `char`-Paaren der erste Wert im Bereich 0xD800–0xDBFF liegt und der zweite im Bereich 0xDC00–0xDFFF. Diese Bereiche werden für einzelne `char`-Werte nicht benutzt.

Mit Java-Version 5 wurden außerdem neue Bibliotheksmethoden eingeführt, die mit `int`-codierten Zeichen arbeiten. Das wirkt sich beispielsweise auf Strings aus, in denen zwei aufeinanderfolgende `char`-Elemente ein einziges Unicodezeichen repräsentieren können. Die Anzahl Bytes, die ein String einnimmt, kann folglich größer als die doppelte Stringlänge sein. Das ist dann der Fall, wenn im String Unicodezeichen aus dem oberen Codebereich vorkommen. ◀

Brückenklassen

Kopplung der Textein- und -ausgabe an Byteströme

Die beiden Klassen `OutputStreamWriter` und `InputStreamReader` verbinden einen Bytestrom mit einem Zeichenstrom und werden deshalb auch als „Brückenklassen“ bezeichnet. Ein `OutputStreamWriter` kapselt einen `OutputStream` in einen `Writer`. Entsprechend kapselt ein `InputStreamReader` einen `InputStream` in einen `Reader`. Die beiden Klassen bilden Textdaten in Bytefolgen ab und benutzen dazu ein Encoding, das im Konstruktor angegeben wird.

Beispiel zur Kontrolle des Encodings

Das folgende Programm `Textfile` schreibt das Wort „Tüt“ in eine Textdatei, deren Name als erstes Kommandozeilenargument angegeben wird. Dazu benutzt es das Encoding, das als zweites Argument genannt ist. Wenn das Programm mit einem beliebigen dritten Argument aufgerufen wird, liest es den Inhalt der Textdatei im angegebenen Encoding und gibt ihn auf dem Bildschirm aus:

```
import java.io.*;

public class Textfile {
    public static void main(String... args) throws IOException {
        String encoding = args[1];
        if(args.length == 2)
            try(OutputStream output = new FileOutputStream(args[0]);
                Writer writer = new OutputStreamWriter(output, encoding);
                PrintWriter pwriter = new PrintWriter(writer)) {
                pwriter.println("Tüt");
            }
        else
            try(InputStream input = new FileInputStream(args[0]);
                Reader reader = new InputStreamReader(input, encoding);
                BufferedReader breader = new BufferedReader(reader)) {
                String line = breader.readLine();
                System.out.println(line);
            }
    }
}
```

Listing 1.39: Dateiausgabe oder -eingabe von Umlauten mit wählbarem Encoding.

Das Programm kann folgendermaßen benutzt werden:

```
$ java Textfile umlaut.txt UTF-8
$ java Textfile umlaut.txt UTF-8 read
Tüt
```

Überprüfung des Encodings

Um den binären Inhalt der Datei sichtbar zu machen, kann das Programm `Hexdump` aufgerufen werden:

```
import java.io.*;
```

```
import static java.lang.System.*;

public class Hexdump {
    public static void main(String... args) throws IOException {
        Textfile.main(args);
        try(InputStream input = new FileInputStream(args[0])) {
            for(int code = input.read(); code >= 0; code = input.read())
                out.printf("%02X ", code);
            out.println();
        }
    }
}
```

Listing 1.40: Ausgabe einer Datei als Folge von hexdezimalen Bytewerten.

Aufrufe mit verschiedenen Encodings liefern die unterschiedlichen Bytedarstellungen des gleichen Textes:

```
$ java Hexdump umlaut.txt UTF-8
54 C3 BC 74 0A
$ java Hexdump umlaut.txt ISO-8859-1
54 FC 74 0A
$ java Hexdump umlaut.txt UTF-16LE
54 00 FC 00 74 00 0A 00
$ java Hexdump umlaut.txt UTF-16BE
00 54 00 FC 00 74 00 0A
$ java Hexdump umlaut.txt UTF-16
FE FF 00 54 00 FC 00 74 00 0A
```

Im nächsten Beispiel wird der Umlaut durch ein Fragezeichen (hexadezimaler Code 3F) ersetzt, weil das Encoding US-ASCII keine Codes über 127 kennt und mit dem Code 252 nichts anfangen kann.

```
$ java Hexdump umlaut.txt US-ASCII
54 3F 74 0A
```

Anders als die Brückenklassen verwenden die Klassen `FileReader` und `FileWriter` immer das Default-Encoding und erlauben keine freie Wahl des Encodings. Der Aufruf

```
new FileReader(...)
```

ist damit eine Kurzform für

```
new OutputStreamReader(new FileOutputStream(...), Charset.defaultCharset())
```

Wenn man ein anderes als das Default-Encoding verwenden möchte, führt kein Weg an den Brückenklassen vorbei.

Encoding von Strings

Unabhängig von der Textein- und -ausgabe können auch Strings in Bytefolgen umgewandelt und daraus erzeugt werden, wie das folgende Beispiel zeigt:

Encoding von Strings in Byte-Arrays und zurück

```
import java.io.*;
import java.util.*;

public class StringBytes {
    public static void main(String... args) throws UnsupportedEncodingException {
        String string = "Tüt";

        byte[] bytes = string.getBytes();
        System.out.println(Arrays.toString(bytes));

        String copy = new String(bytes);
        System.out.println(copy);
    }
}
```

Listing 1.41: Umwandlung zwischen String und Byte-Array mit dem Default-Encoding.

Das Programm verwendet das Default-Encoding, in diesem Beispiel ISO-8859-1. Das Programm gibt daher aus:[73]

```
$ java StringBytes
[84, -4, 116]
Tüt
```

Die Methode `getBytes` und der String-Konstruktor sind jeweils mit einem zusätzlichen Parameter überladen, der ein bestimmtes Encoding auswählt. Erweitert man das Programm `StringBytes` entsprechend, so können auch andere Encodings benutzt werden:

```
import java.io.*;
import java.util.*;

public class StringBytesEncoding {
    public static void main(String... args) throws UnsupportedEncodingException {
        String string = "Tüt";
        String encoding = args[0];

        byte[] bytes = string.getBytes(encoding);
        System.out.println(Arrays.toString(bytes));
```

[73] Das Array enthält `byte`-Elemente. Der primitive Typ `byte` ist vorzeichenbehaftet, deshalb wird der Code von „ü“, 252, als 252 - 256 = -4 wiedergegeben.

```
            String copy = new String(bytes, encoding);
            System.out.println(copy);
        }
    }
```

Listing 1.42: Umwandlung zwischen Strings und Byte-Arrays mit wählbarem Encoding.

Zum Beispiel liefert der Aufruf mit UTF8-Encoding:

```
$ java StringBytes UTF-8
[84, -61, -68, 116]
Tüt
```

1.7 Definition neuer I/O-Klassen

Neue konkrete Byteströme

Die Klassen im Package `java.io` und in den verwandten Packages lassen sich einfach um neue Klassen ergänzen, wie an vorangegangenen Beispielen deutlich wurde. Einen wesentlichen Beitrag zu dieser Offenheit leistet das zugrunde liegende Decorator-Pattern. Dessen Struktur legt zwei Anknüpfungspunkte nahe, nämlich konkrete Elemente und Dekoratoren.

Die Klasse `RandomInputStream` (Listing 1.17) ist ein einfaches Beispiel für einen neuen konkreten `InputStream`.

`NullOutputStream`

`OutputStream`, der alles verwirft

Das vielleicht kürzeste mögliche Beispiel eines konkreten `OutputStream` ist der `NullOutputStream`, der alle Ausgaben verwirft.[74] Ein solches Objekt eignet sich zu Testzwecken, bei denen die Ausgabe keine Rolle spielt, aus technischen Gründen aber ein Abnehmer in Form eines `OutputStream` erforderlich ist.

```
import java.io.*;

public class NullOutputStream extends OutputStream {
    public void write(int code) {
    }
}
```

Listing 1.43: Konkreter `OutputStream`, der die Ausgabe verwirft.

[74] Gegenstücke auf Betriebssystemebene sind die Pseudofiles `/dev/null` von Unix und `nul` von Windows. Alle auf diese Dateien geschriebenen Daten werden einfach gelöscht.

`FailingReader`

Reader, der an einer bestimmten Stelle scheitert

Nützlich zu Testzwecken ist ein konkreter Reader, der nach einer vorgegebenen Anzahl Zeichen mit einer `IOException` scheitert. Welche Zeichen bis zur Exception konkret geliefert werden, spielt weiter keine Rolle. Hier wird willkürlich das `x` gewählt. Die Klasse `FailingReader` führt dazu in einer Objektvariablen Buch über die verbleibende Anzahl `x` bis zur Exception.

In der ABC `Reader` sind die Methoden `close` und `read` mit drei Argumenten abstrakt und müssen hier definiert werden. `close` kann leer bleiben, weil es keine Ressourcen zu bereinigen gibt. Die `read`-Methode unterscheidet die folgenden Fälle:

1. Der Aufrufer verlangt keine Zeichen: In diesem Fall kehrt `read` sofort zurück.
2. Es stehen noch ausreichend viele Zeichen zur Verfügung: Der Puffer wird mit der gewünschten Anzahl Zeichen aufgefüllt und der Zähler entsprechend verringert.
3. Der Vorrat verbleibender Zeichen ist zu klein, aber nicht leer: `read` ruft sich selbst noch einmal mit der noch übrigen Zeichenanzahl auf.
4. Der Vorrat ist aufgebraucht: Jetzt scheitert `read` endgültig mit einer `IOException`.

```
import java.io.*;
import java.util.*;

public class FailingReader extends Reader {
    private int remaining;

    public FailingReader(int remaining) {
        this.remaining = remaining;
    }

    public void close() throws IOException {
    }

    public int read(char[] buffer, int start, int num) throws IOException {
        if(num == 0)
            return 0;
        else if(remaining >= num) {
            remaining -= num;
            Arrays.fill(buffer, start, num, 'x');
            return num;
        }
        else if(remaining > 0)
            return read(buffer, start, remaining);
        else
            throw new IOException("read error");
    }
}
```

Listing 1.44: Reader, der nach einer vorgegebenen Anzahl Zeichen scheitert.

CesarCipherWriter

Modell eines verschlüsselnden Writers

Als Beispiel für einen `FilterWriter` wird eine Textverschlüsselung implementiert, die Sie vielleicht noch aus Kindertagen kennen: Bei der sogenannten Cäsar-Verschlüsselung wird jeder Buchstabe um eine feste Anzahl von Positionen im Alphabet verschoben, wobei nach dem „Z“ wieder mit dem „A“ begonnen wird.

```
import java.io.*;

public class CesarWriter extends FilterWriter {
    private final int offset;

    public CesarWriter(Writer writer, int offset) {
        super(writer);
        this.offset = (offset%26 + 26)%26;
    }

    public void write(int chr) throws IOException {
        if(chr >= 'A' && chr <= 'Z') {
            chr += offset;
            if(chr > 'Z')
                chr -= 26;
        }
        super.write(chr);
    }

}
```

Listing 1.45: Textausgabefilter zur Cäsar-Verschlüsselung.

Zur Entschlüsselung kann der `CesarWriter` mit einer negativen Distanz initialisiert werden.[75]

DetabReader

Auflösen von Tabulatorzeichen in Leerzeichenfolgen

Der Algorithmus zum Auflösen von Tabulatorzeichen in der Klasse `Detab` (Listing 1.4) lässt sich als `FilterReader` zur Verfügung stellen und ist damit allgemein verwendbar:

```
import java.io.*;

public class DetabReader extends FilterReader {
    private final int tabWidth;
```

[75] Der Ausdruck `(offset%26 + 26)%26` implementiert den mathematischen Modulus, der jeden `int`-Wert in den Bereich 0–25 abbildet. Im Gegensatz dazu liefert der Ausdruck `offset%26` bei negativen Werten von `offset` negative Ergebnisse.

```
    private int length = 0;             // aktuelle Zeilenlänge

    private int remaining = 0;          // Anzahl verbleibender Blanks

    public DetabReader(Reader reader, int tabWidth) {
        super(reader);
        this.tabWidth = tabWidth;
    }

    public int read() throws IOException {
        if(remaining > 0) {
            length++;
            remaining--;
            return ' ';
        }
        int code = super.read();
        if(code == '\n') {
            length = 0;
            return code;
        }
        if(code == '\t') {
            remaining = tabWidth - 1 - length%tabWidth;
            length++;
            return ' ';
        }
        length++;
        return code;
    }

}
```

Listing 1.46: Textfilter, der eingelesene Tabulatorzeichen zu Leerzeichen expandiert.

Dieser Reader produziert mehr Zeichen als er von seiner Quelle liest. Genauer gesagt wird jedes Tabulatorzeichen in ein oder mehrere Leerzeichen umgesetzt. Nachdem ein einzelner `read`-Aufruf immer nur ein Zeichen liefern kann, müssen gegebenenfalls weitere Leerzeichen für künftige `read`-Aufrufe vorgemerkt werden. Dazu dient die Objektvariable `remaining`, die die Anzahl der noch auszuliefernden Leerzeichen speichert. Beim Aufruf von `read` wird zuerst `remaining` überprüft und nur dann, wenn keine aufgesparten Leerzeichen mehr übrig sind, tatsächlich das nächste Zeichen von der Quelle geholt und umgesetzt.

`EntabWriter`

Ersatz von Leerzeichenfolgen durch Tabulatorzeichen

Zur Vollständigkeit sei hier auch der Gegenspieler des `DetabReader` (Listing 1.46) gezeigt, ein `EntabWriter`. Er packt Folgen von zwei oder mehr Blanks, die an einer Spaltenposition enden, zu einem Tabulatorzeichen zusammen und verringert damit die Zeichenanzahl.

```
import java.io.*;

public class EntabWriter extends FilterWriter {
    private final int tabWidth;

    private int length = 0;     // aktuelle Zeilenlänge

    private int blanks = 0;     // Anzahl schließender Blanks

    public EntabWriter(Writer writer, int tabWidth) {
        super(writer);
        this.tabWidth = tabWidth;
    }

    public void close() throws IOException {
        flush();
        super.close();
    }

    public void flush() throws IOException {
        for(int i = 0; i < blanks; i++)
            super.write(' ');
        super.flush();
    }

    public void write(int chr) throws IOException {
        if(chr == '\n') {
            for(int i = 0; i < blanks; i++)
                super.write(' ');
            blanks = 0;
            super.write(chr);
            length = 0;
        }
        else if(chr == '\t') {
            super.write(chr);
            blanks = 0;
            length = (length + tabWidth)/tabWidth*tabWidth;
        }
        else if(chr == ' ') {
            blanks++;
            length++;
            if(length%tabWidth == 0) {
                if(blanks > 1)
                    super.write('\t');
                else
                    super.write(' ');
                blanks = 0;
            }
        }
        else {
            for(int i = 0; i < blanks; i++)
                super.write(' ');
            blanks = 0;
            super.write(chr);
            length++;
        }
    }
```

```
}
```

Listing 1.47: Textausgabefilter, der Leerzeichen zu Tabulatorzeichen zusammenfasst.

Dieser Algorithmus gibt alle Zeichen aus, außer den Leerzeichen. Letztere werden zunächst nur mitgezählt. Sobald die Zeilenlänge eine Spaltenbreite erreicht, wird statt einer Folge von zwei oder mehr Leerzeichen nur ein einzelnes Tabulatorzeichen ausgegeben. Diese Klasse gibt weniger Zeichen weiter, als sie annimmt. Hier ist insbesondere die Definition von `flush` und `close` essenziell, weil die Klasse unter Umständen Leerzeichen zurückhält, die beim Schließen des Writers ausgegeben werden müssen.

Bedeutung von `flush` und `close`

1.8 Datei-Operationen und Directories

Die vorangegangenen Abschnitte dieses Kapitels befassen sich mit Ein- und Ausgabe im Allgemeinen, unter anderem auch von und zu Files. Dabei geht es aber ausschließlich um den Zugriff auf den *Inhalt*. In diesem Abschnitt wird die Bearbeitung von Informationen *über* Dateien betrachtet, wie zum Beispiel deren Name, Größe, Zugriffsrechte und Zeitmarken. Dahinter steht der Umgang von Java mit Filesystemen und ihrem Inhalt.

Arbeiten mit Dateien als Ganzes

1.8.1 Datei-Objekte

Die Klasse `File` wird seit Langem benutzt, um mit Filesystem-Elementen zu arbeiten. Mit der Zeit haben sich aber gewisse Unzulänglichkeiten dieser Klasse gezeigt, sodass sie in Java Version 7 von der Klasse `Path` im Package `java.nio.file` abgelöst wurde. `Path` ist Teil des Projekts „New I/O 2“ (NIO2)[76] und arbeitet mit anderen NIO2-Typen zusammen, dabei insbesondere mit der Klasse `Files`. In der Laufzeitbibliothek findet sich NIO2 im Package `java.nio.file` und darunterliegenden Packages.[77]

Klassen `File` und `Path`

`Path` ist ein Interface, das von verschiedenen konkreten Klassen implementiert wird.

Factory-Methode zum Erzeugen von `Path`-Objekten

[76] Der Begriff „New I/O 2“ ist etwas unglücklich gewählt. Die „2“ dient zur Abgrenzung vom älteren Projekt „NIO“, dessen Akronym aber für „Non-blocking I/O“ steht.

[77] Objekte der Typen `File` und `Path` können ineinander umgewandelt und nebeneinander im gleichen Programm benutzt werden. Allerdings umfasst `Path` die gesamte Funktionalität von `File` und noch einiges mehr, sodass der Einsatz von `File` nicht mehr nötig ist.

`Path`-Objekte werden nicht mit Konstruktoren erzeugt, sondern von „Factory-Methoden“[78], die in der Klasse `Paths`[79] definiert sind. Die Anweisung

```
Path path = Paths.get("somefile.txt");
```

erzeugt ein `Path`-Objekt, das die Datei `somefile.txt` repräsentiert.

`Path`-Objekte ohne Bezug zum Filesystem

Die Factory-Methode prüft nicht, ob die angegebene Datei existiert oder ob der Pfadname überhaupt sinnvoll ist. Sie konstruiert nur ein Objekt innerhalb der JVM und greift nicht auf das Filesystem zu. Infolgedessen kann `get` auch nicht mit einer `IOException` scheitern.

Allerdings erkennt `get`, ob der angegebene String nach den Namensregeln des zugrunde liegenden Filesystems grundsätzlich zulässig ist. Wenn der String *keinesfalls* eine Datei repräsentieren kann, bricht die Methode mit einer Exception ab. So führt der Aufruf

```
Path path = Paths.get("\u0000");
```

auf Unix-Systemen zu einer `InvalidPathException`, weil Unix keine 0-Bytes in Dateinamen erlaubt.

1.8.2 Eigenschaften von Dateien

Statische `Files`-Methoden zum Zugriff auf das Filesystem

Ein `Path`-Objekt repräsentiert einen Pfadnamen, der einen Punkt im Filesystem lokalisiert. Eigenschaften des betreffenden Filesystem-Elements lassen sich über statische Methoden der Klasse `Files` ermitteln, die einen `Path` als Argument erwarten. Das folgende Programm `PathInfo` fragt eine Reihe von Informationen ab und gibt sie aus:[80]

```
import java.io.*;
import static java.lang.System.*;
import java.nio.file.*;

public class PathInfo {
    public static void main(String... args) throws IOException {
```

[78] Factory-Methoden sind eine Alternative zu Konstruktoren. Sie werden im Kapitel 8.1 ausführlich diskutiert.

[79] `Paths` ist eine Utility-Klasse mit ausschließlich statischen Methoden. Sie kann weder instanziiert werden, noch wären ihre Objekte sinnvoll verwendbar.

[80] Hier wurde auf eine statische Import-Klausel verzichtet, um die Zuordnung der Methodenaufrufe zur `Files`-Klasse herauszustellen.

```
        Path path = Paths.get(args[0]);
        out.println(Files.exists(path));
        out.println(Files.getLastModifiedTime(path));
        out.println(Files.getOwner(path));
        out.println(Files.size(path));
        out.println(Files.isDirectory(path));
        out.println(Files.isExecutable(path));
        out.println(Files.isHidden(path));
        out.println(Files.isReadable(path));
        out.println(Files.isRegularFile(path));
        out.println(Files.isSymbolicLink(path));
        out.println(Files.isWritable(path));
    }
}
```

Listing 1.48: Ausgabe der Metadaten einer Datei.

Startet man das Programm mit dem eigenen Bytecode, dann erhält man beispielsweise die folgende Ausgabe. Die Kommentare wurden nachträglich eingefügt, um die Informationen dem oben abgedruckten Code zuordnen zu können.

```
$ java PathInfo PathInfo.class
true                    // exists
2011-04-08T05:01:43Z    // getLastModifiedTime
rs                      // getOwner
1549                    // size
false                   // isDirectory
false                   // isExecutable
false                   // isHidden
true                    // isReadable
true                    // isRegularFile
false                   // isSymbolicLink
true                    // isWritable
```

Filesystem-spezifische Ausprägung von Eigenschaften

Die Interpretation einiger Eigenschaften hängt vom zugrunde liegenden Filesystem ab. Deutlich wird das beim Vergleich eines FAT-Filesystems mit einem typischen Unix-Filesystem:

- Die Eigenschaft `isHidden` für „versteckte Files“ wird von FAT durch ein Flag umgesetzt, das jedem File zugewiesen werden kann. In Unix gelten dagegen Dateien, deren Name mit einem Punkt beginnt, automatisch als „versteckt“ und andere nicht.[81]
- Umgekehrt liegen die Verhältnisse bei `isExecutable` für „ausführbare Dateien“, also Programmdateien. FAT betrachtet Files, deren Name auf `exe` oder `com` endet, implizit als ausführbar. Unix verwendet für diese Eigenschaft ein Flag und kein Namensschema.

[81] Das Unix-Kommando `ls` zeigt versteckte Files nur auf Nachfrage, das heißt mit der Option `-a`.

- `isSymbolicLink` ist eine typische Unix-Eigenschaft und hat in FAT kein Gegenstück.

Filesystem-spezifische Attribute

FAT- und Unix-Filesysteme sind zwei populäre Vertreter, die von vielen Systemen unterstützt werden. Die beiden Filesysteme haben unterschiedliche Eigenschaften, die von der oben gezeigten Sammlung von Auskunftsmethoden allerdings etwas grob über einen Kamm geschoren werden.

Attribute populärer Filesysteme

Den Zugriff auf Filesystem-spezifische Eigenschaften verschafft die Methode `readAttributes`. Sie erwartet neben einem `Path` als zweites Argument die Angabe eines Filesystem-Typs in Form eines Typobjekts. Die Laufzeitbibliothek stellt die folgenden vordefinierten Typen zur Verfügung:

`BasicFileAttributes`	Gemeinsame Attribute
`DosFileAttributes`	FAT-Attribute
`PosixFileAttributes`	Unix-Attribute

Das folgende Beispielprogramm `PathAttributes` ruft alle drei Attributsammlungen ab. Während die `BasicFileAttributes` auf jedem System bekannt sind, wird eine der beiden nachfolgenden scheitern.

```
import java.io.*;
import static java.lang.System.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

public class PathAttributes {
    public static void main(String... args) throws IOException {
        Path path = Paths.get(args[0]);

        BasicFileAttributes basicAtts = Files.readAttributes(path, BasicFileAttributes.cl
        out.println(basicAtts.creationTime());
        out.println(basicAtts.isDirectory());
        out.println(basicAtts.isOther());
        out.println(basicAtts.isRegularFile());
        out.println(basicAtts.isSymbolicLink());
        out.println(basicAtts.lastAccessTime());
        out.println(basicAtts.lastModifiedTime());
        out.println(basicAtts.size());

        try {
            DosFileAttributes dosAtts = Files.readAttributes(path, DosFileAttributes.clas
            out.println(dosAtts.isArchive());
            out.println(dosAtts.isHidden());
            out.println(dosAtts.isReadOnly());
            out.println(dosAtts.is));
```

```
        }
        catch(IOException iox) {
            out.println(iox);
        }

        try {
            PosixFileAttributes posixAtts = Files.readAttributes(path, PosixFileAttribute
            out.println(posixAtts.group());
            out.println(posixAtts.owner());
            out.println(posixAtts.permissions());
        }
        catch(IOException iox) {
            out.println(iox);
        }
    }
}
```

Listing 1.49: Ausgabe der Metadaten eines Pfadnamens.

Auf einem Unix-System erhält man beispielsweise die nachfolgende Ausgabe. Der Zugriffsversuch auf FAT-Attribute scheitert dort erwartungsgemäß. Wieder wurden nachträglich Kommentare eingefügt, um die Ausgaben dem obigen Quelltext besser zuordnen zu können:

```
$ java PathAttributes PathAttributes.class
2011-04-08T06:12:51Z            // creationTime
false                           // isDirectory
false                           // isOther
true                            // isRegularFile
false                           // isSymbolicLink
2011-04-08T06:13:08Z            // lastAccessTime
2011-04-08T06:12:51Z            // lastModifiedTime
2341                            // size
java.nio.file.FileSystemException: PathAttributes.class: Operation not supported
users                           // group
rs                              // owner
[OWNER_WRITE, OWNER_READ]       // permissions
```

1.8.3 Zugriffsrechte

Ändern von Metadaten

Ein Teil der Eigenschaften von Dateien kann nicht nur gelesen, sondern auch verändert werden. Für die Länge einer Datei gilt das beispielsweise nicht.[82] Der Umgang mit einigen Eigenschaften erfordert zudem ausreichende Rechte, die das Betriebssystem überwacht. So ist das Ändern des Datei-Besitzers in der Regel nur mit Administratorrechten möglich, über die ein normaler Benutzer aus gutem Grund nicht verfügt.

[82] Natürlich kann die Dateilänge durch Manipulation des Datei-*Inhalts* verändert werden, nicht aber durch bloße Manipulation der Meta-Informationen.

Die Klasse `Files` definiert über die oben gezeigten Auskunftsmethode hinaus auch Methoden zum Ändern von Meta-Informationen. Das folgende Programm demonstriert den Einsatz solcher Methoden.[83] Es erwartet als Kommandozeilenargumente:

- einen `Path`,
- den Namen des Benutzers, dem die Datei übereignet wird,
- eine Anzahl Stunden, um die die letzte Änderung verschoben wird,
- eine Liste von Unix-Zugriffsrechten, die der Datei zugewiesen werden.

```
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.util.*;

public class PathChanges {
    public static void main(String... args) throws IOException {
        PathAttributes.main(args);  // print

        int arg = 0;
        Path path = Paths.get(args[arg++]);
        System.out.println("--- applying changes");

        FileSystem fileSystem = FileSystems.getDefault();
        UserPrincipal owner = fileSystem.getUserPrincipalLookupService().lookupPrincipalB
        path = Files.setOwner(path, owner);

        FileTime time = FileTime.fromMillis(System.currentTimeMillis() + Long.parseLong(a
        path = Files.setLastModifiedTime(path, time);

        Set<PosixFilePermission> perms = new HashSet<>();
        for(String perm: args[arg++].split(","))
            perms.add(PosixFilePermission.valueOf(perm));
        path = Files.setPosixFilePermissions(path, perms);

        PathAttributes.main(args);
    }
}
```

Listing 1.50: Manipulation der Metadaten von Dateien.

Kontrolle erlaubter Änderungen durch das Betriebssystem

Der Aufruf des Programms

```
$ java PathChanges PathChanges.class dummy -24 OWNER_READ,GROUP_READ
...
--- applying changes
```

[83] Die Hilfsklassen `UserPrincipal`, `FileTime` und so weiter stehen hier nicht im Mittelpunkt und werden nicht weiter diskutiert.

```
Exception in thread "main" java.nio.file.FileSystemException:
    PathChanges.class: Operation not permitted
```

als normaler Benutzer scheitert, weil die Rechte zum Übereignen der Datei fehlen.[84] Startet man das Programm dagegen mit Administratorrechten, so werden die verlangten Änderungen durchgeführt:[85]

```
# java PathChanges PathChanges.class dummy -24 OWNER_READ,GROUP_READ
2011-10-15T04:58:36Z
false
false
true
false
2011-10-15T04:59:06Z
2011-10-15T04:58:36Z
2783
java.nio.file.FileSystemException: PathChanges.class: Operation not supported
users
rs
[OWNER_WRITE, OWNER_READ]
--- applying changes
2011-10-14T04:59:13Z
false
false
true
false
2011-10-15T04:59:06Z
2011-10-14T04:59:13Z
2783
java.nio.file.FileSystemException: PathChanges.class: Operation not supported
users
dummy
[OWNER_READ, GROUP_READ]
```

1.8.4 Pfade

Arten von Pfaden in einem hierarchischen Filesystem

Moderne Filesysteme sind durchweg hierarchisch organisiert.[86] Die wichtigsten Bewohner sind geschachtelte Directories und Files.[87] Ein **Pfad** im Filesystem benennt ein Element. Er besteht aus einer Liste von Directories, gefolgt vom Namen

[84] Davon ausgenommen ist der wenig sinnvolle Fall, dass sich ein Benutzer eine Datei aneignet, die ihm bereits gehört.

[85] In diesem Beispiel wird angenommen, dass das System den Benutzer „dummy“ kennt.

[86] Windows verwendet aus historischen Gründen noch „Laufwerke“, die mit einzelnen Zeichen, in der Regel Großbuchstaben, benannt sind. Auf jedem Laufwerk liegt ein eigenes, hierarchisches Filesystem.

[87] Je nach Filesystem-Typ kann es noch einen ganzen Zoo von anderen, zum Teil exotischen Elementen geben, die weder Directories noch Files sind.

des Elements selbst. Absolute Pfade beginnen im Root-Directory, relative Pfade beziehen sich auf ein beliebiges Directory als Ausgangspunkt. Hier einige Beispiele:

`/home/rs/PathChanges.java`
: Absoluter Pfad zur Datei `PathChanges.java` im Directory `/home/rs`.

`PathChanges.java`
: Relativer Pfad zur Datei `PathChanges.java` im aktuellen Directory.

`rs/PathChanges.java`
: Relativer Pfad zur Datei `PathChanges.java` beginnend im aktuellen Directory.

Die folgenden `Path`-Methoden machen die einzelnen Elemente eines Pfads zugänglich:

`Path getFileName()`
: Liefert das letzte Element eines Pfads.

`Path getParent()`
: Liefert das Eltern-Directory.

`Path getRoot()`
: Liefert das Root-Directory.

Sonderstellung des Root-Directory

Das Root-Directory selbst hat weder einen Namen, noch ein Eltern-Directory. Die beiden ersten Methoden liefern daher beim Root-Directory `null`. Das folgende Programm `PathElements` zeigt den Einsatz dieser Methoden.

```
import java.nio.file.*;

public class PathElements {
    public static void main(String... args) {
        Path path = Paths.get(args[0]);
        System.out.println(path.getFileName());
        System.out.println(path.getParent());
        System.out.println(path.getRoot());
    }
}
```

Listing 1.51: Ausgabe der Bestandteile eines Pfadnamens.

Ein Aufruf des Programms liefert die folgende Ausgabe:

```
$ java PathElements /home/rs/PathElements.java
PathElements.java
/home/rs
/
```

Diese Methoden greifen nicht auf das Filesystem zu, sondern arbeiten rein lexikalisch, wie das folgende Beispiel mit einem nicht existierenden Pfad zeigt:

Lexikalischer Umgang mit Pfaden

```
$ java PathElements /sim/sala/bim
bim
/sim/sala
/
```

Einzelne Pfadelemente

Ein Pfad besteht aus einer Liste von Elementen. Die einzelnen Elemente können über einen Zugriff per Index oder durch Auslesen eines Iterators erreicht werden.

Aufbau von Pfaden aus Elementen

`int getNameCount()`
: Liefert die Anzahl der Elemente in einem Pfad.

`Path getName(int index)`
: Liefert das Element am gegebenen Index.

Darüber hinaus ist `Path` eine Iterable seiner Elemente, wie das folgende Programm zeigt. Eine *foreach*-Schleife ist bequemer, aber weniger flexibel als eine `for`-Schleife.[88]

```
import java.nio.file.*;

public class PathElementList {
    public static void main(String... args) {
        Path path = Paths.get(args[0]);

        int numElements = path.getNameCount();
        System.out.println(numElements);
        for(int i = 0; i < numElements; i++)
            System.out.println(path.getName(i));

        for(Path element: path)
            System.out.println(element);
```

[88] Zum Beispiel ist die Reihenfolge von vorne nach hinten, aus der Sicht eines Filesystems von der Wurzel nach unten, vorgegeben.

```
    }
}
```

Listing 1.52: Zerlegt einen Pfad in Einzelteile und gibt sie aus.

Es produziert zweimal nacheinander die gleiche Liste:

```
$ java PathElementList /home/rs/PathElementList.java
home
rs
PathElementList.java
home
rs
PathElementList.java
```

Gekürzte, absolute und relative Pfade

Umbau von Pfaden

Die beiden Namen `.` und `..` sind für das aktuelle Directory selbst beziehungsweise für das übergeordnete oder „Eltern-Directory“ reserviert.[89] Der Pfad `/home/rs/PathRelations.java` ist beispielsweise äquivalent zu:

```
/home/./rs/PathRelations.java
/home/./././rs/./././rs/PathRelations.java
/home/rs/../../home/rs/PathRelations.java
```

Solche unnötig verschlungenen Pfade wird kaum ein Mensch vorsätzlich konstruieren, sie können aber durchaus bei automatischen Berechnungen entstehen. Mit den folgenden `Path`-Methoden bekommt man Pfade dieser Art in den Griff:

`Path normalize()`
: Kürzt einen Pfad auf die Mindestlänge durch Löschen überzähliger Elemente `.` und `...`

`Path toAbsolutePath()`
: Liefert einen absoluten Pfad.

`Path toRealPath()`
: Liefert einen absoluten Pfad und löst dabei unterschiedliche Zugriffswege auf. Das betrifft auf FAT-Filesystemen zum Beispiel Groß- und Kleinschreibung in Namen, auf Unix-Filesystemen symbolische Links.

Operationen mit potenziellen Exceptions

Beachten Sie, dass die letzte Methode, anders als die anderen, auf das Filesystem

[89] Das heißt, dass in *jedem* Directory `.` existiert, ein Synonym für das Directory selbst. Abgesehen vom Root-Directory gibt es in jedem Directory auch `..` als Repräsentant des jeweiligen Eltern-Directory.

zugreift. Sie kann daher mit einer `IOException` scheitern.

Das nächste Programm zeigt die Arbeitsweise:

```
import java.io.*;
import java.nio.file.*;

public class PathAbsolute {
    public static void main(String... args) throws IOException {
        Path path = Paths.get(args[0]);
        System.out.println(path.normalize());
        System.out.println(path.toAbsolutePath());
        System.out.println(path.toRealPath());
    }
}
```

Listing 1.53: Variationen von Pfaden.

Ein Aufruf liefert folgendes Ergebnis:

```
$ cd /home
$ java -cp rs PathAbsolute rs/./PathAbsolute.class
rs/PathAbsolute.class
/home/rs/./PathAbsolute.class
/home/rs/PathAbsolute.class
```

1.8.5 Directories

Auslesen von Directories über `DirectoryStream`-Objekte

Den Inhalt von Dateien machen die verschiedenen File-I/O-Klassen im Package `java.io` zugänglich. Die Elemente von Directories lassen sich über `DirectoryStream` Objekte auslesen, die die statische Methode `newDirectoryStream` der Klasse `Files` liefert. Das folgende Programm listet den Inhalt eines Directory auf:

```
import java.io.*;
import java.nio.file.*;

public class ListDir {
    public static void main(String... args) throws IOException {
        Path dirpath = Paths.get(args[0]);
        try(DirectoryStream<Path> dirstream = Files.newDirectoryStream(dirpath)) {
            for(Path path: dirstream)
                System.out.println(path);
        }
    }
}
```

Listing 1.54: Ausgabe des Inhaltes eines Directory mit einem `DirectoryStream`-Objekt.

Es kann folgendermaßen verwendet werden:

```
$ java ListDir .
MakePaths.java
FilesystemProviders.java
PathInfo.java
ListDir.java
PathChanges.java
PathAttributes.java
PathRelations.java
PathAbsolute.java
PathElements.java
```

`Directory-Stream`-Objekte als Ressourcen

`DirectoryStream`-Objekte öffnen ein Directory und belegen dazu Betriebssystem-Ressourcen, wie geöffnete Files. Sie müssen daher ebenso geschlossen werden. Wie die Filestreams implementiert `DirectoryStream` das Interface `Closeable` und wird daher sinnvollerweise mit ARM verwendet, wie im vorhergehenden Beispiel gezeigt.

Eine überladene Version von `newDirectoryStream` akzeptiert als zweites Argument ein Namensmuster und liefert nur Elemente, die zu diesem Muster passen. So erfasst beispielsweise der Directorystream

```
Files.newDirectoryStream(path, "P*.java")
```

nur Elemente, die mit `P` beginnen und auf `.java` enden. Vorsicht: Die Schreibweise der Namensmuster interpretiert die Wildcard-Zeichen `?` und `*` wie eine Shell, folgt aber nicht der Java-Syntax regulärer Ausdrücke!

Rekursiver Directorydurchlauf

Rekursiver Durchlauf eines Directorybaums

Die `Files`-Methode `walkFileTree` vereinfacht den tiefen (rekursiven) Durchlauf eines Directorybaums. Sie erwartet ein Directory als Startpunkt und ein weiteres Objekt vom Typ `FileVisitor`, dessen Methoden beim Durchlauf aufgerufen werden. Solche Methoden werden auch als „Callback-Methoden“ bezeichnet, weil sie nicht direkt aus dem eigenen Code, sondern mittelbar aufgerufen werden.

Callback von `FileVisitor`-Methoden

Das generische Interface `FileVisitor<T>` definiert die folgenden Methoden:[90]

`FileVisitResult preVisitDirectory(T dir, BasicFileAttributes atts)`
: Wird beim Erreichen eines Directory aufgerufen.

[90] Als Typargument für `T` kommt hier nur `Path` infrage.

`FileVisitResult postVisitDirectory(T dir, IOException iox)`
: Wird beim Verlassen eines Directory aufgerufen. Wenn es keine Probleme gab, ist `iox == null`. Andernfalls liefert `ios` Einzelheiten zu den Ursachen der Schwierigkeiten.

`FileVisitResult visitFile(T file, BasicFileAttributes atts)`
: Wird mit jedem File aufgerufen.

`FileVisitResult visitFileFailed(T file, IOException iox)`
: Wird mit jedem Element aufgerufen, das nicht besucht werden kann. Dafür gibt es verschiedene Gründe, wie zum Beispiel mangelnde Zugriffsrechte.
Details liefert das Exceptionobjekt `iox`.

Steuerung des Durchlaufs mit dem Callback-Ergebnis

Der Rückgabewert vom Aufzählungstyp `FileVisitResult` legt fest, wie `walkFileTree` weiter verfährt. Beim Ergebnis `FileVisitResult.CONTINUE` wird der Durchlauf fortgesetzt, bei `FileVisitResult.TERMINATE` abgebrochen.

Beispiel: Suche kurzer Dateien

Das folgende Programm listet alle Dateien auf, die kürzer als 1 kB sind. Bei Problemen bricht das Programm den Durchlauf ab.

```
import java.io.*;
import java.nio.file.*;
import static java.nio.file.FileVisitResult.*;
import java.nio.file.attribute.*;

public class FindShortFiles {
    public static void main(String... args) throws IOException {
        Path path = Paths.get(args[0]);
        Files.walkFileTree(path, new FileVisitor<Path>() {
            public FileVisitResult visitFile(Path file, BasicFileAttributes atts) throws
                if(Files.size(file) < 1 << 10)
                    System.out.println(file);
                return CONTINUE;
            }

            public FileVisitResult postVisitDirectory(Path dir, IOException iox) {
                return CONTINUE;
            }

            public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes iox) {
                return CONTINUE;
            }

            public FileVisitResult visitFileFailed(Path file, IOException iox) throws IOE
                throw iox;
            }
        });
    }
}
```

Listing 1.55: Filevisitor zur rekursiven Suche nach kurzen Dateien in einem Directorybaum.

Vereinfachung mit SimpleFileVisitor

Die vordefinierte Klasse `SimpleFileVisitor<T>` vereinfacht die Implementierung neuer `FileVisitor`-Klassen. Sie besucht alle Files, ohne etwas damit zu unternehmen. Neue Visitor-Klassen müssen nur noch die tatsächlich gebrauchten Methoden redefinieren. Das folgende Programm `FindShortFilesSimple` liefert die gleichen Ergebnisse wie `FindShortFiles` (Listing 1.55), redefiniert aber nur die eine interessante Methode `visitFile` und erbt die Default-Implementierungen der anderen drei Methoden von der Basisklasse `SimpleFileVisitor`:

```
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

public class FindShortFilesSimple {
    public static void main(String... args) throws IOException {
        Path path = Paths.get(args[0]);
        Files.walkFileTree(path, new SimpleFileVisitor<Path>() {
            public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws
                if(Files.size(file) < 1 << 10)
                    System.out.println(file);
                return FileVisitResult.CONTINUE;
            }
        });
    }
}
```

Listing 1.56: Erweiterung des `SimpleFileVisitor` zur Suche nach kurzen Dateien.

1.8.6 Schreiben und Lesen

Schreiben und Lesen des kompletten Datei-Inhalts

Die Klasse `Files` bietet bequeme Methoden an, um mit wenig Aufwand eine Binär- oder Textdatei zu lesen oder zu schreiben. Der Inhalt einer Binärdatei wird dabei in ein Byte-Array abgebildet, der Inhalt einer Textdatei in eine String-Liste.

`byte[] readAllBytes(Path path)`
: Liest den Inhalt von `path` in ein neues Byte-Array und liefert es zurück.

`Path write(Path path, byte[] bytes)`
: Schreibt das Byte-Array `bytes` auf `path` und liefert das erste Argument zurück.

`List<String> readAllLines(Path path, Charset charset)`
: Liest den Inhalt von `path` zeilenweise in eine neu allozierte String-Liste und liefert diese zurück.

`Path write(Path path, Iterable<String> strings, Charset charset)`
: Schreibt die Strings auf `path` und fügt nach jedem String einen Zeilenwechsel an. Liefert das erste Argument zurück.

Die Datei wird von allen Methoden nach dem Lesen oder Schreiben wieder geschlossen, ebenso wie im Fehlerfall.

Die beiden String-bezogenen Methoden verwenden das Encoding gemäß `Charset`-Argument. In einfachen Fällen reicht hier das Argument `Charset.defaultCharset()` für das systemspezifische Default-Encoding aus. Die Klasse `Charset` wurde auf Seite 85 vorgestellt.

Die beiden `write`-Methoden ersetzen den Inhalt einer bereits existierenden Datei. Wenn die Datei verlängert werden soll, kann als weiteres Argument `StandardOpenOption.APPEND` angefügt werden.

Das folgende Programm kopiert eine Datei binär:[91]

```
import java.io.*;
import java.nio.file.*;

public class CopyFileNIO2 {
    public static void main(String... args) throws IOException {
        Path from = Paths.get(args[0]);
        Path to = Paths.get(args[1]);
        Files.write(to, Files.readAllBytes(from));
    }
}
```

Listing 1.57: Kopieren einer Datei mit minimalem Aufwand.

Das nächste Beispielprogramm kopiert eine Textdatei mit dem Default-Encoding und versieht dabei alle Zeilen mit einer Zeilennummer. Es erfüllt den gleichen Zweck wie `LineNumbers` (Listing 1.34):

```
import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;

public class LineNumbersNIO2 {
    public static void main(String... args) throws IOException {
        Path from = Paths.get(args[0]);
        Path to = Paths.get(args[1]);
        Charset charset = Charset.defaultCharset();
        List<String> lines = new ArrayList<>(Files.readAllLines(from, charset));
        for(int i = 0; i < lines.size(); i++)
            lines.set(i, i + 1 + "\t" + lines.get(i));
        Files.write(to, lines, charset);
    }
```

[91] Dieses Programm kann eine Datei ohne Schaden auf sich selbst kopieren, weil das Original zuerst komplett eingelesen und die Kopie erst dann geschrieben wird.

```
}
```

Listing 1.58: Kopiert eine Textdatei und versieht sie dabei mit Zeilennummern.

Einschränkung auf Dateien, die komplett in den Speicher passen

Diese `readAll`- und `write`-Methoden sind kein allgemeiner Ersatz für die übrigen I/O-Mechanismen, die in diesem Kapitel vorgestellt wurden. Die ganze Ein- und Ausgabe zielt auf die Verarbeitung potenziell großer oder gar endloser Datenmengen, während diese Methoden davon ausgehen, dass der Inhalt von Dateien komplett in die JVM passt. Sie eignen sich aber als schnelle Lösung für einfache Anwendungsfälle.

Factory-Methoden für I/O-Klassen

I/O-Klassen über `Path`-Objekte

Die Klasse `Files` bietet einen Satz Factory-Methoden für I/O-Objekte, die von bereits existierenden `Path`-Objekten ausgehen. Sie dienen als Alternativen zu den Konstruktoren der bereits diskutierten konkreten `File`-Klassen:

`InputStream newInputStream(Path path)`
: Liefert einen neuen `InputStream` zum Lesen von Bytes von der Datei `path`.

`OutputStream newOutputStream(Path path)`
: Liefert einen neuen `OutputStream` zum Schreiben von Bytes auf die Datei `path`.

`BufferedReader newBufferedReader(Path path, Charset charset)`
: Liefert einen neuen `BufferedReader` zum Einlesen von Textdaten im angegebenen Encoding.

`BufferedWriter newBufferedWriter(Path path, Charset charset)`
: Liefert einen neuen `BufferedWriter` zum Schreiben von Textdaten im angegebenen Encoding.

Die Rückgabewerte der Methoden sind geöffnete I/O-Objekte, die nach der Verwendung geschlossen werden müssen. Sie sollten als Ressourcen dem ARM untergeordnet werden. Das folgende Programm entspricht dem Programm `FileCopy` (Listing 1.13), benutzt aber Factory-Methoden statt `FileStream`-Konstruktoren:

```
import java.io.*;
import java.nio.file.*;
import static java.nio.file.StandardOpenOption.*;
```

```
public class FileCopyPaths {
    public static void main(String... args) throws IOException {
        Path from = Paths.get(args[0]);
        Path to = Paths.get(args[1]);
        OpenOption option = args.length > 2? APPEND: CREATE;
        try(InputStream input = Files.newInputStream(from);
            OutputStream output = Files.newOutputStream(to, option)) {
            for(int code = input.read(); code >= 0; code = input.read())
                output.write(code);
        }
    }
}
```

Listing 1.59: Einsatz von Factory-Methoden für I/O-Klassen.

1.8.7 Filesystem-Operationen

Grundlegende Operationen mit kompletten Dateien

Operationen mit kompletten Dateien, wie zum Beispiel Umbenennen, Kopieren, Verschieben und Löschen, lassen sich mit `Files`-Methoden auf der Grundlage von `Path`-Objekten abwickeln. Diese Operationen können aus vielen Gründen scheitern und selbst im Erfolgsfall größere Datenmengen bewegen und dabei spürbar Zeit verbrauchen.

`Path move(Path from, Path to)`
: Benennt eine Datei um oder verschiebt sie, wenn Quelle und Ziel in verschiedenen Directories liegen. Liefert `to` als Ergebnis zurück. Die Zieldatei darf nicht existieren. Mit einem optionalen dritten Argument `StandardCopyOption.REPLACE_EXI` wird eine existierende Zieldatei ersetzt.

`Path copy(Path from, Path to)`
: Kopiert eine Datei und liefert den Ziel-`Path` als Ergebnis zurück. Ein drittes Argument steuert das Verhalten bei einer existierenden Zieldatei, wie bei `move`.

`void delete(Path path)`
: Löscht eine Datei.

Das Programm `FileOps` erwartet als erstes Kommandozeilenargument einen Methodennamen, als zweites Argument eine Quelldatei und als optionales drittes Argument eine Zieldatei:

```
import java.io.*;
import java.nio.file.*;

public class FileOps {
    public static void main(String... args) throws IOException {
```

```
            Path from = Paths.get(args[1]);
            Path to = args.length > 1? Paths.get(args[1]): null;
            switch(args[0]) {
                case "move":
                    Files.move(from, to);
                    break;
                case "copy":
                    Files.copy(from, to);
                    break;
                case "delete":
                    Files.delete(from);
                    break;
            }
        }
    }
```

Listing 1.60: Programm für grundlegende Datei-Operationen.

Erzeugen von Filesystem-Elementen zu `Path`-Objekten

`Path`-Objekte repräsentieren hypothetische Filesystem-Elemente. Die folgenden Methoden erzeugen reale Elemente:

`Path createFile(Path path)`
: Erzeugt eine neue, leere Datei in einem Directory, das bereits existiert. Die Datei darf es noch nicht geben.

`Path createDirectories(Path path)`
: Erzeugt ein neues Directory mit allen Eltern-Directories, soweit sie noch nicht existieren.

`Path createTempFile(String prefix, String suffix)`
: Erzeugt eine neue, leere Datei in einem systemspezifischen Directory für temporäre Daten. Die Datei erhält einen Namen, den keine andere Datei in diesem Directory hat.
Der Name der Datei kann mit den Argumenten `prefix` und `suffix` beeinflusst werden. Beide dürfen `null` sein, um die Wahl dem System zu überlassen.

Das Beispielprogramm `FileCreate` erzeugt abhängig vom ersten Kommandozeilenargument eine neue Datei, ein neues Directory oder eine temporäre Datei.

```
import java.io.*;
import java.nio.file.*;

public class FileCreate {
    public static void main(String... args) throws IOException {
        Path path = args.length > 1? Paths.get(args[1]): null;
        switch(args[0]) {
            case "file":
```

```
                Files.createFile(path);
                break;
            case "directories":
                Files.createDirectories(path);
                break;
            case "tempfile":
                path = Files.createTempFile(null, null);
                break;
        }
        System.out.println(path);
    }
}
```

Listing 1.61: Erzeugt gemäß Kommandozeilenargument eine neue Datei, ein neues Directory oder eine temporäre Datei.

Eine temporäre Datei kann beispielsweise folgendermaßen erzeugt werden. Jeder Aufruf liefert eine andere Datei:

```
$ java FileCreate tempfile
/tmp/3249489288236961685.tmp
$ java FileCreate tempfile
/tmp/3895520753245104699.tmp
```

Kopieren von Byteströmen von und auf Dateien

Schließlich kann mit einer überladenen `copy`-Methode ein `InputStream` komplett in eine Datei geschrieben werden. Eine weitere `copy`-Methode überträgt den Inhalt einer Datei komplett in einen `OutputStream`.

`long copy(InputStream from, Path to)`
: Kopiert den `InputStream` bis zum Ende in die Datei `to`.

`long copy(Path from, OutputStream to)`
: Kopiert den Inhalt der Datei `from` komplett auf den `OutputStream`.

Das folgende Programm kopiert den Inhalt einer Datei auf die Standardausgabe:

```
import java.io.*;
import java.nio.file.*;

public class ShowFile {
    public static void main(String... args) throws IOException {
        Files.copy(Paths.get(args[0]), System.out);
    }
}
```

Listing 1.62: Kopieren des Inhaltes einer Datei auf die Standardausgabe.

Zusammenfassung

- Die **Standardein- und -ausgabe** verläuft über die Methoden `read` beziehungsweise `write` der Objekte `System.in`, `System.out` und `System.err`.
- `read` signalisiert das **Eingabeende** mit dem **Fluchtwert** -1.
- Das Betriebssystem kann die Standardein- und -ausgabe von und zu **Dateien umlenken**.
- Die **Standard-Fehlerausgabe** (`System.err`) umgeht die reguläre Ausgabe-Umlenkung.
- Die ABCs `InputStream` und `OutputStream` bilden die **Grundlage der Ein- und Ausgabe** von Byteströmen.
- Ausgaben sind **gepuffert**, die Methode `flush` entleert den Puffer.
- Alle Streams müssen mit `close` **geschlossen** werden.
- **ARM-Blöcke** (*Automatic Resource Management*, auch als *try with resource* bezeichnet) stellen `close`-Aufrufe sicher.
- Das ARM baut auf den **Interfaces** `Closeable` **und** `AutoCloseable`.
- **Filestreams** (`FileInputStream` und `FileOutputStream`) implementieren die ABCs und erlauben das **Lesen und Schreiben von Dateien**.
- Mit Byte-Arrays überladene `read`- und `write`-Methoden übertragen effizient **Byteblöcke**.
- **Filterklassen transformieren** „durchfließende“ Daten und können zu Filterketten verknüpft werden.
- Filterklassen referenzieren einen anderen Stream und sind selbst Streams. Sie setzen das **Decorator-Pattern** um.
- Die ABCs `Reader` und `Writer` sind die Basis für **Text-I/O.** Darunter liegt eine Klassenhierarchie, die symmetrisch zu den Byteströmen aufgebaut ist.
- Die **Brückenklassen** `InputStreamReader` und `OutputStreamWriter` implementieren das **Encoding** von Text in Binärdaten und zurück.
- Die Klassen im Package `java.io` sind offen für die Erweiterung um **neue Streams**.
- `Path`-Objekte repräsentieren **Dateien im Filesystem**.
- Die Klasse `Files` definiert statische Methoden zum Umgang mit **Metadaten** (Zugriffsrechte, Besitzer, Zeitmarken und so weiter).
- Die Klassen `Files` und `DirectoryStream` ermöglichen programmgesteuerte **Datei-Operationen** (Kopieren, Löschen, Verschieben und ähnliche) im hierarchischen Filesystem.

Aufgaben

Aufgabe 1: Streams mit Methodenaufruf-Protokoll

Die tatsächlichen Methodenaufrufe an einen Stream sind bei einer Konstruktion mit Filtern nicht immer leicht zu überschauen. Definieren Sie zwei Filterklassen `TracingInputStream` und `TracingOutputStream`, die alle Methodenaufrufe unverändert an den zugrunde liegenden Stream delegieren, die Aufrufe aber auf einem zusätzlichen `OutputStream` protokollieren.

Die Konstruktoren der beiden Klassen haben die folgenden Signaturen:

`TracingInputStream(InputStream source, String key, OutputStream log)`
: Protokolliert alle Methodenaufrufe auf dem Stream `log`. Jede Zeile beginnt mit dem String `key`.

`TracingOutputStream(OutputStream sink, String key, OutputStream log)`
: Entsprechend zum vorhergehenden Konstruktor.

Die folgende Beispielanwendung schaltet vor und nach einen `BufferedOutputStream` jeweils einen `TracingOutputStream`:

```
import java.io.*;
import java.util.*;

public class TracingOutputStreamMain {
    public static void main(String... args) throws IOException {
        ByteArrayOutputStream bytesOutput = new ByteArrayOutputStream();
        try(TracingOutputStream tracingOutput = new TracingOutputStream(bytesOutput, "Byt
            BufferedOutputStream bufferedOutput = new BufferedOutputStream(tracingOutput)
            TracingOutputStream tracingBufferedOutput = new TracingOutputStream(buffered0
            for(String arg: args)
                tracingBufferedOutput.write(arg.getBytes());
        }
        System.out.println(Arrays.toString(bytesOutput.toByteArray()));
    }
}
```

Listing 1.63: Überprüfung der Pufferung durch einen `BufferedOutputStream`.

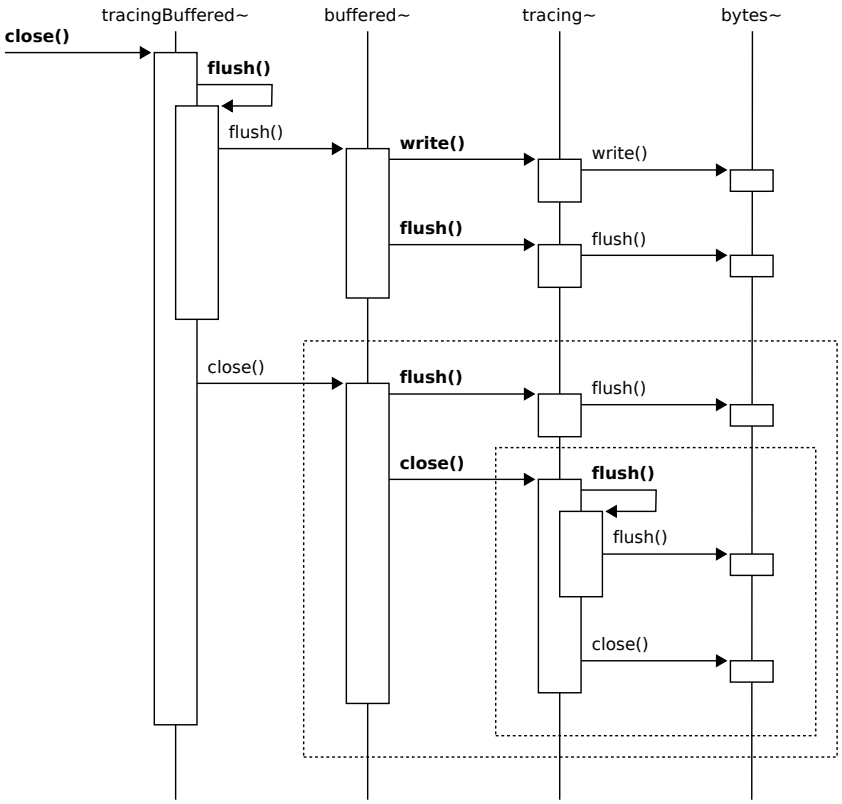
Startet man das Programm mit ein paar Kommandozeilenargumenten, so zeigt sich die Wirkungsweise der Pufferung:

```
$ java TracingOutputStreamMain abra ka dabra
```

```
Buffered write(byte[4])
Buffered write(byte[4], 0, 4)
Buffered write(byte[2])
Buffered write(byte[2], 0, 2)
Buffered write(byte[5])
Buffered write(byte[5], 0, 5)
Buffered close()
Buffered flush()
ByteArray write(byte[8192], 0, 11)
ByteArray flush()
ByteArray flush()
ByteArray close()
ByteArray flush()
ByteArray flush()
ByteArray close()
ByteArray flush()
ByteArray close()
ByteArray flush()
[97, 98, 114, 97, 107, 97, 100, 97, 98, 114, 97]
```

Es lässt sich gut erkennen, dass sich die einzelnen Ausgaben von `main` (Zeilen `Buffered write`...) in einer einzigen Ausgabe des zugrunde liegenden Streams (Zeile `ByteArray write`...) sammeln, wobei die Größe des Puffers in diesem Beispiel 8192 Bytes (8 kB) beträgt.

Auf den ersten Blick überraschend wirkt vielleicht die Kaskade der `flush`- und `close`-Aufrufe am Ende. Sie ergeben sich aus dem Abschluss des ARM-Blocks, der die im Kopf definierten Streams in umgekehrter Reihenfolge nacheinander schließt. Die `close`-Aufrufe lösen ihrerseits zuerst ein `flush`, dann ein `close` des zugrunde liegenden Streams aus. Wegen der geforderten Idempotenz bleiben die mehrfachen Aufrufe ohne Folgen. Die folgende Skizze zeigt den Ablauf des ersten `close`-Aufrufs des Objekts `tracingBufferedOutput`. Der größere eingerahmte Ausschnitt folgt im Anschluss noch einmal beim Schließen von `bufferedOutput`, daraufhin noch der kleinere eingerahmte Ausschnitt bei `tracingOutput`:



Die fett gedruckten Aufrufe richten sich an die beiden `TracingOutputStream`-Objekte. Sie erscheinen im oben wiedergegebenen Protokoll. Die übrigen Aufrufe laufen unsichtbar ab.

Aufgabe 2: Ausgabe mit Rücknahme

Die Bibliotheksklasse `PushbackInputStream` (siehe Seite 62) kann das zuletzt gelesene Byte in den Eingabe-Bytestrom „zurückgeben“, sodass es noch einmal gelesen werden kann. Entwickeln Sie eine neue Filterklasse `UnwriteOutputStream`, die das zuletzt geschriebene Byte zurücknehmen kann. Dazu definiert die Klasse die zusätzliche Methode

```
int  unwrite()
```

die die Ausgabe des letzten Bytes ungeschehen macht und dieses Byte zurückliefert. Vor der ersten Ausgabe ist das Ergebnis -1. `unwrite` kann nur ein einziges

Byte zurücknehmen. Wiederholte Aufrufe ohne zwischenzeitliche Ausgabe liefern ebenfalls -1 und haben sonst keine Wirkung.

Natürlich rückt ein normaler Ausgabe-Bytestrom einmal geschriebene Daten nicht mehr heraus. Deshalb hält `UnwriteOutputStream` „auf Verdacht“ immer das letzte Byte zurück, für den Fall, dass die Ausgabe rückgängig gemacht werden soll. Anschaulich gesagt hinkt die tatsächliche Ausgabe auf den zugrunde liegenden `OutputStream` der Ausgabe auf den `UnwriteOutputStream` immer um ein Byte hinterher.

Als problematisch erweist sich das Verhalten von `flush`. Diese Methode soll generell *alle* irgendwo gepufferten Daten ausgeben, wonach aber `unwrite` nichts mehr zurücknehmen kann. `UnwriteOutputStream` gibt der „höheren“ Aufgabe den Vorrang: `flush` übermittelt ein möglicherweise zurückgehaltenes Byte zum zugrunde liegenden Ausgabe-Bytestrom. Ein anschließendes `unwrite` liefert -1, so als hätte es keine vorangegangene Ausgabe gegeben.

Zum Testen eignet sich das folgende Programm:

```
import java.io.*;
import java.util.*;

public class UnwriteOutputStreamMain {
    public static void main(String[] args) throws IOException {
        ByteArrayOutputStream bytesOutput = new ByteArrayOutputStream();
        try(UnwriteOutputStream uos = new UnwriteOutputStream(bytesOutput)) {
            for(String arg: args)
                if("u".equals(arg))
                    System.out.println("unwrite: " + uos.unwrite());
                else if("f".equals(arg))
                    uos.flush();

                else
                    uos.write(Integer.parseInt(arg));
        }
        System.out.println(Arrays.toString(bytesOutput.toByteArray()));
    }
}
```

Listing 1.64: Testprogramm für `UnwriteOutputStream`.

Es holt Bytewerte von der Kommandozeile und schreibt sie in einen `ByteArrayOutputStream`, dessen Inhalt am Ende auf dem Bildschirm gezeigt wird. Die Buchstaben `f` und `u` zwischen den Bytewerten lösen Aufrufe von `flush` und `unwrite` aus. Das Programm kann beispielsweise folgendermaßen verwendet werden:

```
$ java UnwriteOutputStreamMain 1 2 3 u u 4 f u 5 6 7 u
```

```
3
-1
-1
7
[1, 2, 4, 5, 6]
```

Sollten sich Probleme auftun, so nutzen Sie die Lösung der vorhergehenden Aufgabe zum Protokollieren von Aufrufen.

Aufgabe 3: Streams mit natürlichen Zahlen

Streams transportieren Bytewerte, mit denen sich Zahlen im Bereich von 0 bis 255 direkt codieren lassen. Für größere Zahlen sind mehrere Bytes nötig. Unter der Annahme, dass *die meisten* Werte unter 256 liegen, aber einige wenige darüber, steht man vor dem Dilemma, dass ein paar Ausreißer zur Codierung *aller* Werte in längeren Bytefolgen zwingen. Eine Codierung von Zahlen in einer *variablen* Anzahl Bytes mildert das Problem.[92]

Diese Aufgabe klammert negative Werte aus, weil sich daraus diffizile Probleme beim Umgang mit dem Ende der Eingabe ergeben.

Definieren Sie zwei Filterstream-Klassen `IntegerOutputStream` und `IntegerInputStream`, die zusammenarbeiten. Die Methode `write(int)` von `IntegerOutputStream` schreibt für ein nicht negatives Argument n die folgenden Bytes auf den zugrunde liegenden `OutputStream`:

- 1 bis 255: ein Byte mit dem Wert n.
- 256 bis 65535 (2^8 bis $2^{16} - 1$): ein 0-Byte, dann zwei Bytes für n.
- 65536 und mehr (2^{16} bis $2^{31} - 1$): zwei 0-Bytes, dann vier Bytes für n.
- 0: sechs 0-Bytes.

Die Methode `int read()` der Klasse `IntegerInputStream` decodiert diese Bytefolgen und liefert als Ergebnis den entsprechenden Zahlenwert im Bereich von 0 bis $2^{31} - 1$. Zahlen werden also, abhängig von ihrer Größe, in 1, 3 oder 6 Bytes ausgegeben.

Die `IntegerOutputStream`-Methoden zum Schreiben von Byte-Arrays erfordern bei geschickter Implementierung der Klasse keinen zusätzlichen Aufwand. Heikler sind die Methoden zum Lesen von Byte-Arrays, weil sie nicht alle Daten eines

[92] Ein ähnliches Problem zeigt sich bei Strings: Zeichen im ASCII-Zeichensatz können in einzelnen Bytes codiert werden, aber andere Zeichen belegen mehrere Bytes. Einen Kompromiss weisen die UTF-Codierungen mit variabler Byteanzahl pro Zeichen.

`IntegerInputStream` aufnehmen können. Lassen Sie sie eine `UnsupportedOperationException` werfen, weil keine adäquate Implementierung möglich ist.

Zur Lösung dieser Aufgabe ist die genaue Abbildung von Zahlen in Bytes entscheidend. Ein 16-Bit-Wert zerfällt in zwei Bytes, ein niederwertiges mit den Bits 0–7 und ein höherwertiges mit den Bits 8–15. Das **Little-Endian-Format** beginnt mit dem niederwertigen Byte, gefolgt vom höherwertigen. Im **Big-Endian-Format** stehen die Bytes genau andersherum. Die erste Zeile im folgenden Beispiel zeigt die Codierung von n = 256 = 0x100 im Little-Endian-Format, die zweite im Big-Endian-Format:

```
0, 0, 1
0, 1, 0
```

Ein `IntegerInputStream` würde die erste Zeile allerdings falsch interpretieren, weil er nach den beiden 0-Bytes *vier* weitere Bytes erwartet und nicht nur ein einziges! Deshalb *müssen* die `IntegerStream`-Klassen mit dem Big-Endian-Format arbeiten.

Die folgende Testanwendung erwartet eine Reihe von Zahlen auf der Kommandozeile. Sie schreibt diese Zahlen über einen `IntegerOutputStream` in ein Byte-Array. Anschließend liest sie wieder Zahlen aus dem Byte-Array und gibt sie aus:

```
import java.io.*;

public class IntegerStreamMain {
    public static void main(String... args) throws IOException {
        ByteArrayOutputStream byteArrayOutput = new ByteArrayOutputStream();
        try(IntegerOutputStream integerOutput = new IntegerOutputStream(byteArrayOutput))
            for(String arg: args)
                integerOutput.write(Integer.parseInt(arg));
        }

        ByteArrayInputStream byteArrayInput = new ByteArrayInputStream(byteArrayOutput.to
        try(IntegerInputStream integerInput = new IntegerInputStream(byteArrayInput)) {
            int n = integerInput.read();
            while(n >= 0) {
                System.out.println(n);
                n = integerInput.read();
            }
        }
    }
}
```

Listing 1.65: Testanwendung für die Klassen `IntegerInputStream` und `IntegerOutputStream`.

Ein Aufruf sollte die Zahlen der Kommandozeile unverändert wieder ausgeben:

```
$ java IntegerStreamMain 0 1 2 255 256 65535 65536 2000000000
```

```
0
1
2
255
256
65535
65536
2000000000
```

Aufgabe 4: Stream-Iterator

Definieren Sie eine Klasse `StreamBytes`, die den Inhalt eines beliebigen `InputStream` als `Iterable<Byte>` zur Verfügung stellt. Ein Fluchtwert für das Eingabeende ist hier nicht erforderlich, deshalb kann mit Bytes gearbeitet werden. Für eine halbwegs einfache Implementierung sind Einschränkungen sinnvoll:

- Anders als allgemeine `Iterable`-Objekte darf ein `StreamBytes`-Objekt nur einmal durchlaufen werden. Beim Versuch, ein Objekt in zweites Mal zu durchlaufen, wird eine `IllegalStateException` geworfen.
- Ein `InputStream`, der einem `StreamBytes`-Objekt übergeben wurde, darf auf keinem anderen Weg verändert werden. Die `StreamBytes`-Klasse kann das allerdings nicht aus eigener Kraft sicherstellen.
- Die `Iterator`-Methode `remove` wirft eine `UnsupportedOperationException`, weil Änderungen eines `InputStream` auf diesem Weg zur Vereinfachung nicht vorgesehen wird.

Das folgende Beispielprogramm zeigt den Einsatz von `StreamBytes`:

```
import java.io.*;
import java.util.*;

public class StreamBytesMain {
    public static void main(String... args) throws IOException {
        try(InputStream input = new FileInputStream(args[0])) {
            for(byte code: new StreamBytes(input))
                System.out.println(code);
        }
    }
}
```

Listing 1.66: Testprogramm für `StreamBytes`-Iterables.

Der folgende zusätzliche Code überprüft, ob sich ein `StreamBytes`-Objekt tatsächlich nur einmal durchlaufen lässt:

```
StreamBytes streamBytes = new StreamBytes(input);
Iterator<Byte> iterator = streamBytes.iterator();   // erster Iterator
try {
    iterator = streamBytes.iterator();              // zweiter Iterator
    throw new AssertionError("should have failed before!");
}
catch(IllegalStateException ex) {
    // o.k.
}
```

Listing 1.67: Nur ein Iterator pro Stream erlaubt.

Aufgabe 5: File-Vergleich

Definieren Sie eine Klasse `FileComparator`, die das Interface `Comparator<Path>` implementiert, zum Vergleich von Files. Dabei gelten die folgenden Kriterien:

- Wenn eine Datei nicht existiert, wirft der Comparator eine `IllegalArgumentException`.
- Wenn ein Argument kein normales File ist, wirft der Comparator eine `IllegalArgumentException`.
- Wenn beim Lesen eines Files ein Fehler auftritt, wirft der Comparator eine `RuntimeException`, die die auslösende `IOException` kapselt.
- Ansonsten entscheidet das erste abweichende Byte über die Reihenfolge: Die Datei mit dem kleineren Bytewert wird vorne einsortiert. Im folgenden Beispiel gilt die erste Datei als „kleiner“, weil sie im ersten Paar abweichender Bytes das kleinere enthält:

  ```
  1, 2, 3,  4, 5, 6, 7
  1, 2, 3,  5, 6, 7
  ```

- Wenn die längere Datei mit dem ganzen Inhalt der kürzeren Datei beginnt, wird die kürzere Datei vorne einsortiert. Im folgenden Beispiel gilt die erste Datei als „kleiner“, weil ihr ganzer Inhalt Präfix der zweiten Datei ist:

  ```
  1, 2, 3
  1, 2, 3, 5, 6, 7
  ```

- Folglich sind nur Dateien mit gleicher Länge und gleichem Inhalt als Ganzes „gleich“.

Das folgende Programm liest Dateinamen von der Kommandozeile und sortiert die entsprechenden Dateien mit einem `FileComparator`. Es gibt die sortierte Liste anschließend wieder aus:

```
import java.nio.file.*;
import java.util.*;
```

```
public class FileComparatorMain {
    public static void main(String[] args) {
        List<Path> list = new ArrayList<>();
        for(String arg: args)
            list.add(Paths.get(arg));
        Collections.sort(list, new FileComparator());
        System.out.println(list);
    }
}
```

Listing 1.68: Testprogramm für einen `FileComparator`.

Die Klasse soll Dateien *nicht* komplett in den Speicher laden, sondern nur so weit wie nötig lesen. Mit dem folgenden Programm lassen sich passende Testdateien erzeugen, um die geforderte Eigenschaft nachzuweisen. `WriteBinaryFile` erwartet Kommandozeilenargumente der Form *numxcode* und gibt eine Datei aus, in der jeweils *num* Bytes mit dem Wert *code* aufeinanderfolgen.

```
public class WriteBinaryFile {
    public static void main(String... args) {
        for(String arg: args) {
            String[] token = arg.split("x");
            int num = Integer.parseInt(token[0]);
            int code = Integer.parseInt(token[1]);

            while(num-- > 0)
                System.out.write(code);
        }
        System.out.flush();
    }
}
```

Listing 1.69: Programm zum Erzeugen von Test-Eingabedateien für den `FileComparator`.

Erzeugen Sie ein Paar von Dateien, die mit den Bytes 1 beziehungsweise 2 beginnen, jeweils gefolgt von zehn Millionen 0-Bytes. Die beiden Dateien lassen sich schnell vergleichen:

```
$ java WriteBinaryFile 1x1 10000000x0 > 10...
$ java WriteBinaryFile 1x2 10000000x0 > 20...
$ time java FileComparatorMain 10... 20...
[10..., 20...]
real    0m0.417s
```

Wenn die abweichenden Bytes am Ende stehen, dauert der Vergleich viel länger:

```
$ java WriteBinaryFile 10000000x0 1x1 > 0...1
```

```
$ java WriteBinaryFile 10000000x0 1x2 > 0...2
$ time java FileComparatorMain 0...1 0...2
[0...1, 0...2]
real    1m57.513s
```

Kapitel

2 Serialisierung

Lernziele

In diesem Kapitel lernen Sie

- wie ein Java-Programm Objekte in eine binäre Datei **serialisieren** und später wieder daraus **rekonstruieren** kann, sodass Objekte Programmstarts überleben oder von einer JVM in eine andere übertragen werden können.
- wie Sie mit **JavaBeans** ein ähnliches Ziel erreichen, die Objekte aber in einem XML-Dokument gespeichert werden. Ein XML-Dokument kann man lesen, ausdrucken und sogar mit einem Texteditor bearbeiten.
- die externe Bibliothek **XStream** kennen, die einige Vorteile der beiden vorhergehenden Ansätze kombiniert.

Die in den Variablen eines Java-Programms gespeicherten Daten existieren nur so lang, wie das Programm läuft. Wenn das Programm endet, gehen die Variablenwerte unwiederbringlich verloren. Der Begriff **Persistenz** fasst Verfahren zusammen, mit denen Datenstrukturen über Programmläufe hinweg erhalten werden können. Persistenz kann auf verschiedenen Wegen erreicht werden: Das sogenannte OR-Mapping bildet Java-Objekte in relationale Datenbanken ab, während Serialisierung Objekte im Filesystem speichert. Letzteres ist der Gegenstand dieses Kapitels.

Mit Serialisierung lassen sich Datenstrukturen nicht nur auf Datenträgern sichern, sondern auch zwischen zwei verschiedenen, gleichzeitig laufenden Java-Programmen austauschen. Diese beiden Programme können in unterschiedlichen JVMs auf demselben Rechner laufen oder auf getrennten, mit einem Netzwerk verbundenen Rechnern.[1] Konzeptionell bieten Serialisierungsmechanismen die Möglichkeit, Datenstrukturen zwischen *zeitlich* oder *räumlich* getrennten JVMs zu übermitteln.

[1] Letztlich können Datenstrukturen sogar in ein und demselben Programm konserviert und sofort wieder rekonstruiert werden. Das Ergebnis ist ein Clone, das heißt eine identische, aber unabhängige Kopie der ursprünglichen Daten.

Die Ein- und Ausgabe (Kapitel 1) erlaubt das Schreiben und Lesen von Bytes und Textzeichen auf und von Files. Files speichern aber nur flache Folgen, während Programme mit komplexen, hochstrukturierten Daten arbeiten. Grundsätzlich lassen sich natürlich alle Strukturen irgendwie in Bytes und Textzeichen gießen.[2] Die dazu nötige **Serialisierung** und **Deserialisierung** (Restaurierung) ist allerdings nicht einfach. Dieses Problem lösen die Verfahren, die in diesem Kapitel vorgestellt werden, auf verschiedene Arten.[3]

Bei allen Serialisierungsmechanismen muss der Anwender mehr oder weniger Unterstützung leisten, sei es durch Einsatz bestimmter Bibliotheksmethoden, durch Einhaltung von Vorgaben oder durch ergänzenden eigenen Code. Es gibt keine Vollautomatik, die ohne jedes Zutun auf „magische Art" die Datenstrukturen eines Java-Programms festhält und zu einem anderen Zeitpunkt oder in einer anderen JVM wieder zur Verfügung stellt.

2.1 Fragen

Allgemeine Probleme der Serialisierung

Bevor einzelne Serialisierungsmechanismen im Detail untersucht werden, sollen ein paar grundsätzliche Fragen angesprochen werden, mit denen sich jedes Vorgehen auseinandersetzen muss.

Ziel: Objekte zwischen JVMs transportieren

Ein Serialisierungsverfahren stellt die gleichen Objekte wieder zur Verfügung, die früher existierten oder die in einer anderen JVM leben. Die restaurierten Objekte sollen von den Originalen nicht unterscheidbar sein.[4]

Abhängige Objekte, Objektgraphen

Objekte existieren selten isoliert, sondern referenzieren sich gegenseitig. Zusammen mit einem Objekt müssen also auch alle weiteren Objekte gesichert werden, auf die dieses Objekt direkt oder indirekt Bezug nimmt. Im Allgemeinen hat man es mit Objektgraphen zu tun, die komplett zu serialisieren sind. Das schließt die `null`-Referenz, Selbstreferenzen und Referenzzyklen mit ein.

Nicht serialisierbare Daten

Serialisierungsmechanismen sollten mit möglichst allen Daten zurechtkommen. Unproblematisch sind primitive Werte und Objekte, die komplett innerhalb der JVM

2 Der Hauptspeicher, in dem die Datenstrukturen leben, besteht nur aus Bytes.

3 Man könnte das ganze Thema Serialisierung als Teil von I/O betrachten. Die Schwierigkeiten ergeben sich aber weniger aus der eigentlichen Ein- und Ausgabe selbst als vielmehr aus konzeptionellen Fragen, wie beispielsweise der Behandlung von Referenzzyklen, Bytecode-Versionen und Objekt-Identitäten. Aus diesem Grund ist der Serialisierung dieses eigene Kapitel gewidmet.

4 Es gibt unvermeidliche Abweichungen. Beispielsweise liefert die `Object`-Methode `toString` einen String, in den Hashcode des Objekts eingeht. Der Hashcode beruht wiederum auf der Speicheradresse, die bei der Serialisierung nicht erhalten bleibt. Andererseits ist diese Abweichung nicht revelant, weil der konkrete Adresswert für Benutzercode keine Bedeutung hat.

leben und in sich abgeschlossen sind. Problematisch sind dagegen Daten, die mit Ressourcen außerhalb der JVM zusammenhängen oder an Verwaltungsinformationen der JVM gekoppelt sind, wie zum Beispiel Streams, Threads und Sockets.

Sonderbehandlung von Klassenvariablen

Klassenvariablen sind unabhängig von einzelnen Objekten. Sie können in mancher Hinsicht als Teil des Codes aufgefasst werden und erfordern daher eine andere Behandlung als reguläre Objektvariablen. Das betrifft beispielsweise Aufzählungswerte und Singletons, das heißt Klassen, von denen nur ein einziges Objekt existiert.

Ausnahmen von der Serialisierung

Obwohl die Serialisierungsmechanismen mit den meisten Objekten gut zurechtkommen, kennen sie die Intentionen des Entwicklers nicht. Zum Beispiel ist es nicht immer sinnvoll, blind alle Bits ud Bytes eines Objekts zu sichern und zu restaurieren, weil manche Informationen redundant oder nur für die aktuelle JVM von Bedeutung sind. Serialisierungsmechanismen müssen daher Ausnahmen zulassen.

Umgang mit Konstruktoraufrufen

Die Frage nach Konstruktoraufrufen führt zu widersprüchlichen Anforderungen. Einerseits wurden serialisierte Objekte zu einer anderen Zeit oder in einer anderen JVM geschaffen, sodass bei der Deserialisierung keine neuen Konstruktoraufrufe angebracht sind. Andererseits bedeutet das, dass Objekte *ohne Konstruktoraufrufe* auf dem Heap auftauchen.[5] Es gibt keine allgemeine Lösung dieses Problems. Die verschiedenen Serialisierungsmechanismen gehen unterschiedlich damit um.

Evolution der Klassendefinitionen

Serialisierte Objekte überdauern möglicherweise längere Zeit. In dieser Zeit können sich die ursprünglichen Klassendefinitionen der serialisierten Objekte ändern. Ausschlaggend sind sicher Art und Umfang der Veränderungen. Ein zusätzliches `public` vor einer Methodendefinition sollte das Restaurieren serialisierter Objekte eigentlich nicht verhindern. Eine Umbenennung von Objektvariablen würde das Einlesen allerdings zum Ratespiel machen, sodass die Deserialisierung am besten ganz verweigert werden sollte. Wieder gibt es keine scharfe Grenze.

Dieser Katalog lässt sich fortsetzen. Der Rest des Kapitels stellt Serialisierungsmechanismen mit sehr unterschiedlichen Eigenschaften vor, die sich aber alle mit den hier aufgezählten Fragen auseinandersetzen müssen.

2.2 Object-Streams

I/O-Filterklassen zur Serialisierung

Die abstrakten Basisklassen `InputStream` und `OutputStream` (Seite 28) repräsentieren allgemeine, flache Byteströme. Filterklassen (Seite 1.4) transformieren Bytes *in transit* auf unterschiedlichste Art. Die beiden Filterklassen `ObjectOutputStream` und `ObjectInputStream` brauchen, wie alle Filter, einen anderen Bytestrom als Abnehmer beziehungsweise Lieferanten von Bytes.

[5] Auch die nicht ganz unproblematische Methode `clone` produziert Objekte ohne Konstruktoraufruf.

2.2.1 Methoden `readObject` und `writeObject`

Methoden zum Lesen und Schreiben von Objekten

`ObjectOutputStream` und `ObjectInputStream` serialisieren Java-Objekte in Bytefolgen beziehungsweise deserialisieren (rekonstruieren) Objekte wieder daraus. Sie erfüllen damit einen ähnlichen Zweck wie die beiden Filterklassen `DataOutputStream` und `DataInputStream` (Seite 64). Letztere können allerdings nur mit primitiven Werten und Strings umgehen, während die Object-Streams fast beliebige Objekte verarbeiten.[6] Die wichtigsten Methoden von `ObjectOutputStream` und `ObjectInputStream` sind:

`void writeObject(Object x)`
: Serialisiert ein Objekt in eine Bytedarstellung.

`Object readObject()`
: Rekonstruiert ein Objekt aus einer Bytedarstellung.

`readObject` liefert einen Wert des statischen Typs `Object`. Das tatsächlich rekonstruierte Objekt hat natürlich den gleichen Typ wie das ursprünglich ausgegebene Objekt. Hier führt kein Weg an einem Typecast zurück zum dynamischen Typ vorbei.[7]

Beispiele serialisierter Objekte

Das folgende Programm erwartet einen Dateinamen als Kommandozeilenargument. Das Programm erzeugt ein `Date`- und ein `Random`-Objekt und serialisiert sie nacheinander auf die Datei. Wenn ein beliebiges zusätzliches Kommandozeilenargument angegeben ist, deserialisiert das Programm die beiden Objekte wieder von der Datei.

```
import java.io.*;
import java.util.*;

public class ObjectIO {
    public static void main(String... args) throws IOException, ClassNotFoundException {
        if(args.length == 1)
            // output
            try(OutputStream output = new FileOutputStream(args[0]);
                ObjectOutputStream objectOutput = new ObjectOutputStream(output)) {
                objectOutput.writeObject(new Date());
                objectOutput.writeObject(new Random());
            }
        else
            // input
```

[6] `ObjectOutputStream` und `ObjectInputStream` implementieren die allgemeineren Interfaces `DataOutput` und `DataInput`. Das Gleiche gilt für `DataOutputStream` und `DataInputStream`. Auf dieser Ebene sind die `ObjectStream`-Klassen austauschbar mit den `DataStream`-Klassen.

[7] Ein Typecast auf den falschen Typ löst eine `ClassCastException` aus.

```
            try(InputStream input = new FileInputStream(args[0]);
                ObjectInputStream objectInput = new ObjectInputStream(input)) {
                System.out.println(objectInput.readObject());
                Random random = (Random)objectInput.readObject();
                System.out.println(random.nextInt());
            }
        }
    }
```

Listing 2.1: Ein- und Ausgabe von Objekten mit `ObjectInputStream` und `ObjectOutputStream`.

Rekonstruktion ohne Konstruktoraufruf

Der Zufallszahlengenerator im `Random`-Objekt wird im Konstruktor mit einem Wert initialisiert, in den die aktuelle Systemzeit mit hoher Auflösung einfließt. Beim Deserialisieren wird jedes Mal das *gleiche* Objekt rekonstruiert, deshalb liefert der Aufruf von `nextInt` jedes Mal denselben Wert:

```
$ java ObjectIO objects.ser
$ java ObjectIO objects.ser read
Thu Oct 13 09:05:34 CEST 2011
1579696329
$ java ObjectIO objects.ser read
Thu Oct 13 09:05:34 CEST 2011
1579696329
```

Dieses Beispiel zeigt, dass beim Deserialisieren kein neues Objekt mit einem neuen Konstruktoraufruf erzeugt wird, sondern dass ein früher erzeugtes Objekt rekonstruiert wird.

2.2.2 Interface `Serializable`

Markierung serialisierbarer Klassen

Nicht bei allen Java-Objekten ist Serialisierung überhaupt sinnvoll oder wünschenswert.[8] Aus diesem Grund muss eine Klassendefinition als Ganzes ausdrücklich als „geeignet zur Serialisierung“ ausgewiesen werden. Unterbleibt diese Erklärung, so lassen sich die Objekte nicht serialisieren.

Tagging Interface `Serializable`

Eine Klasse wird serialisierbar, wenn sie das Interface `java.io.Serializable` implementiert. Dieses Interface enthält keine Methoden, sondern wirkt nur durch seine Existenz. Es wird deshalb als „Tagging Interface“ oder „Marker-Interface“ bezeichnet.

[8] Objekte können schützenswerte Informationen enthalten, wie Passwörter, Kontonummern und medizinische Angaben. Gleiches gilt für flüchtige Daten, wie Zeitstempel, Netzwerkverbindungen oder eine Mauszeiger-Position. Beliebige Objekte sollten also besser nicht pauschal serialisierbar sein.

Die Bibliotheksklassen `Date` und `Random` implementieren das Interface `Serializable`. Das Beispielprogramm `ObjectIO` (Listing 2.1) kann diese Objekte daher serialisieren und restaurieren. Das Gleiche gilt für die meisten anderen Bibliotheksklassen. Ausgenommen sind Klassen, die sich nicht sinnvoll rekonstruieren lassen, wie zum Beispiel I/O-Klassen.

Die folgende Klasse `Counter` repräsentiert einen Zähler, der ab 0 in Einerschritten mit `step` hochgezählt und mit `read` abgelesen werden kann.

```
public class Counter {
    private int value = 0;

    public Counter() {
        System.out.println("Counter()");
    }

    public int read() {
        return value;
    }

    public Counter step() {
        value++;
        return this;
    }
    public String toString() {
        return getClass().getSimpleName() + "@" + value;
    }
}
```

Listing 2.2: Zählerklasse.

Das Programm `CounterIO` entscheidet aufgrund der Anzahl der Kommandozeilenargumente, ob es

1. einen `Counter` erzeugt, benutzt und serialisiert, oder
2. einen `Counter` deserialisiert und ausgibt.

```
import java.io.*;

public class CounterIO {
    public static void main(String... args) throws IOException, ClassNotFoundException {
        if(args.length == 1)
            try(OutputStream output = new FileOutputStream(args[0]);
                ObjectOutputStream objectOutput = new ObjectOutputStream(output)) {
                Counter counter = new Counter().step().step();
                objectOutput.writeObject(counter);
            }
        else
            try(InputStream input = new FileInputStream(args[0]);
```

```
                ObjectInputStream objectInput = new ObjectInputStream(input)) {
                Counter counter = (Counter)objectInput.readObject();
                System.out.println(counter);
            }
    }
}
```

Listing 2.3: Ein- und Ausgabe von Objekten via Object-Streams.

Fehlerbericht bei nicht serialisierbaren Objekte

Der Versuch, ein `Counter`-Objekt zu serialisieren, führt zunächst zu einer `NotSerializableException`, weil das Interface `Serializable` fehlt. Wenn man die Definition von `Counter` (Listing 2.2) um die Implementierung von `Serializable` zu

```
public class Counter implements Serializable { ...
```

ergänzt (und dieses Interface importiert), dann gelingen die Serialisierung und die Deserialisierung mit dem Programm `CounterIO` (Listing 2.3):

```
$ java CounterIO objects.ser
$ java CounterIO objects.ser read
Counter@2
$ java CounterIO objects.ser read
Counter@2
```

2.2.3 Objektgraphen

Serialisierbare Bausteine

Object-Streams kommen mit Referenzzyklen, Selbst- und `null`-Referenzen zurecht.

Die folgende Klasse `DualCounter` kapselt ein Paar von zwei getrennten Zählern, die einzeln hochgezählt werden können:

```
import java.io.*;

public class DualCounter implements Serializable {
    private final Counter first;

    private final Counter second;

    public DualCounter(Counter first, Counter second) {
        System.out.printf("DualCounter(%s, %s)%n", first, second);
        this.first = first;
        this.second = second;
    }

    public DualCounter step(int which) {
        (which == 1? first: second).step();
```

```
        return this;
    }

    public int max() {
        return Math.max(first.read(), second.read());
    }

    public String toString() {
        return "DualCounter{" + first + ";" + second + "}";
    }
}
```

Listing 2.4: Serialisierbare Klasse mit Referenzen auf andere Objekte.

`DualCounter` implementiert ebenfalls das Interface `Serializable` und kann folglich serialisiert werden. Mit einem `DualCounter`-Objekt werden auch dessen Objektvariablen, das heißt, die beiden referenzierten `Counter`-Objekte ausgegeben.

Für eine erfolgreiche Serialisierung müssen *alle* Objekte im Graphen das Interface `Serializable` implementieren, sonst kommt es zu einer `NotSerializableException`.

Serialisierung von Arrays

Auch Arrays implementieren das Interface `Serializable`. Sie sind serialisierbar, wenn der Elementtyp serialisierbar ist.

2.2.4 Stream-Struktur

Bausteine von Object-Streams

Ein Object-Stream besteht aus einer Sequenz von serialisierten Objekten und primitiven Werten in beliebiger Abfolge. Es gibt keine einfache Möglichkeit, die Struktur eines Object-Streams zu ergründen. Eine Anwendung muss also „wissen“, welche und wie viele Werte und Objekte sie von einem Object-Stream lesen kann. Liest sie über das Ende hinaus, so wird eine `EOFException` geworfen. Es gibt keinen Fluchtwert, der das Ende eines Object-Streams signalisiert, wie bei normalen I/O-Klassen.

Das folgende Programm serialisiert eine Anzahl einzelner `Counter` und fügt nach dem letzten Objekt den Wert `null` als „Prellbock“ an. Die konkrete Anzahl der ausgegebenen Objekte liegt nicht fest. Sie ergibt sich aus der Sekundenzahl der Systemzeit beim Programmaufruf.

```
import java.io.*;

public class CounterListIO {
    public static void main(String... args) throws IOException, ClassNotFoundException {
        if(args.length == 1)
            try(OutputStream output = new FileOutputStream(args[0]);
                ObjectOutputStream objectOutput = new ObjectOutputStream(output)) {
```

```
                long seconds = System.currentTimeMillis()/1000%60;
                for(int i = 0; i < seconds; i++)
                    objectOutput.writeObject(new Counter());
                objectOutput.writeObject(null);
                System.out.println(seconds + " objects written");
            }
        else
            try(InputStream input = new FileInputStream(args[0]);
                ObjectInputStream objectInput = new ObjectInputStream(input)) {
                int numObjects = 0;
                Counter counter = (Counter)objectInput.readObject();
                while(counter != null) {
                    numObjects++;
                    counter = (Counter)objectInput.readObject();
                }
                System.out.println(numObjects + " objects read");
            }
    }
}
```

Listing 2.5: Serialisierung einer Liste von Objekten mit Ende-Markierung.

Das Programm erkennt beim Einlesen am Wert `null` das Ende der Objektfolge:

```
$ java CounterListIO objects.ser
48 objects written
$ java CounterListIO objects.ser read
48 objects read
$ java CounterListIO objects.ser
0 objects written
$ java CounterListIO objects.ser read
0 objects read
```

Eine derartige Markierung ist nicht die einzige Möglichkeit zum Steuern der Deserialisierung. Alternativ hätte die Folge beispielsweise mit einem primitiven Zählerwert eingeleitet werden können. In jedem Fall ist es aber Sache der Anwendung selbst, die Struktur der Objektfolge passend zu arrangieren.

2.2.5 Aufruf von Konstruktoren

Deserialisierung ohne Konstruktoraufrufe

Bei der Deserialisierung werden Objekte aus dem Bytestrom restauriert, *ohne* dass Konstruktoren aufgerufen werden. Die Objekte existieren ja bereits und waren bloß zeitweise in einem Object-Stream „eingefroren“. Sie wurden irgendwann früher, möglicherweise in einer anderen JVM, durch Aufruf von Konstruktoren erzeugt.

Das folgende Programm serialisiert beziehungsweise deserialisiert einen `DualCounter` (Listing 2.4):

```
import java.io.*;

public class DualCounterIO {
    public static void main(String... args) throws IOException, ClassNotFoundException {
        if(args.length == 1)
            try(OutputStream output = new FileOutputStream(args[0]);
                ObjectOutputStream objectOutput = new ObjectOutputStream(output)) {
                DualCounter counter = new DualCounter(new Counter(), new Counter());
                counter.step(1).step(2).step(1);
                System.out.println(counter);
                objectOutput.writeObject(counter);
            }
        else
            try(InputStream input = new FileInputStream(args[0]);
                ObjectInputStream objectInput = new ObjectInputStream(input)) {
                DualCounter counter = (DualCounter)objectInput.readObject();
                System.out.println(counter);
            }
    }
}
```

Listing 2.6: Deserialisierung von Objekten ohne Konstruktoraufruf.

Definiert man in den Klassen `Counter` und `DualCounter` Konstruktoren, die ihren Aufruf protokollieren, dann erhält man beim Serialisieren die folgende Ausgabe:

```
$ java DualCounterIO objects.ser
Counter()
Counter()
DualCounter(Counter@0, Counter@0)
DualCounter{Counter@2;Counter@1}
```

Beim Deserialisieren werden die Konstruktoren nicht aufgerufen, die entsprechenden Ausgaben bleiben aus:

```
$ java DualCounterIO objects.ser read
DualCounter{Counter@2;Counter@1}
```

Nicht serialisierbare Basisklasse

Eine weitere Frage ergibt sich bei Vererbung: Eine abgeleitete Klasse kann das Interface `Serializable` implementieren, während dass das für die Basisklasse nicht zutrifft.[9] Die folgende Klasse `JumpCounter` erweitert die Zählerklasse um eine neue Methode `jump`, die den Zähler um mehrere Schritte auf einmal hochzählt.

```
import java.io.*;
```

[9] `Object` implementiert beispielsweise *nicht* das Interface `Serializable`.

```
public class JumpCounter extends Counter implements Serializable {
    public JumpCounter() {
        System.out.println("JumpCounter()");
    }

    public JumpCounter jump(int n) {
        while(n-- > 0)
            step();
        return this;
    }
}
```

Listing 2.7: Serialisierbare Klasse mit nicht serialisierbarer Basisklasse.

Benutzt man als Basisklasse die ursprüngliche Fassung von `Counter` (Listing 2.2), die das Interface `Serializable` *nicht* implementiert, so lässt sich ein `JumpCounter` dennoch serialisieren. Allerdings zeigt sich, dass beim Deserialisieren zwar das abgeleitete Objekt ohne Konstruktoraufruf restauriert wird, aber nicht das Basisklassenobjekt. Dieses wird stattdessen mit einem regulären Aufruf des Default-Konstruktors neu erzeugt.

Aufruf von Default-Konstruktoren nicht serialisierbarer Basisklassen

Diese Regelung ist eine Notlösung, die zu einem schwer nachvollziehbaren Verhalten führen kann. Ändert man das Programm `DualCounterIO` (Listing 2.6) so ab, dass der `DualCounter` mit `JumpCountern` initialisiert wird

```
DualCounter dcounter = new DualCounter(new JumpCounter(), new JumpCounter());
```

dann zeigt sich das Problem beim Deserialisieren:

```
$ java DualCounterIO objects.ser
Counter()
JumpCounter()
Counter()
JumpCounter()
DualCounter(JumpCounter@0, JumpCounter@0)
DualCounter{JumpCounter@2;JumpCounter@1}
$ java DualCounterIO objects.ser read
Counter()
Counter()
DualCounter{JumpCounter@0;JumpCounter@0}
```

Die Ausgabe zeigt die Aufrufe der Basisklassenkonstruktoren. Die beiden restaurierten `JumpCounter` (Listing 2.7) starten hier wieder mit 0 und nicht mit den ursprünglichen Werten 2 und 1, die bei der vorhergehenden Serialisierung gegolten haben.

Laufzeitfehler bei fehlendem Default-Konstruktor

Die Deserialisierung ruft *immer* Default-Konstruktoren auf, um nicht serialisierbare

Basisklassenobjekte zu erzeugen. Falls es in einer solchen Basisklasse überhaupt keinen Default-Konstruktor gibt, scheitert das Deserialisieren zur Laufzeit mit einer `InvalidClassException`.

In der umgekehrten Richtung ergibt sich dieses Problem nicht. Eine serialisierbare Basisklasse vererbt die Serialisierbarkeit zwangsläufig an alle abgeleiteten Klassen.

Für reibungslose Serialisierung sollten also alle Klassen im Objektgraphen, einschließlich der Basisklassen, das Interface `Serializable` implementieren.

2.2.6 Cloning

Duplizieren von Objekten über Byteströme

Object-Streams erlauben eine interessante Duplizierung von Objektgraphen. Als Filterstreams können `ObjectInput-` und `-OutputStream` nicht nur mit Files arbeiten, sondern auch mit beliebigen I/O-Klassen. Verwendet man `ByteArrayStreams`, dann verlässt das serialisierte Objekt die JVM nicht. Das folgende Programm `CounterClones` serialisiert einen `JumpCounter` (Listing 2.7) in ein Byte-Array und deserialisiert das Array sofort wieder in einen neuen `JumpCounter`. Dieser zweite `JumpCounter` ist ein Clone des Originals, er entsteht ohne Konstruktoraufruf!

```
import java.io.*;

public class CounterClones {
    public static void main(String... args) throws IOException, ClassNotFoundException {
        byte[] bytes = null;
        JumpCounter counter = new JumpCounter().jump(3).jump(2);
        try(ByteArrayOutputStream bytesOutput = new ByteArrayOutputStream();
            ObjectOutputStream objectOutput = new ObjectOutputStream(bytesOutput)) {
            objectOutput.writeObject(counter);
            objectOutput.flush();
            bytes = bytesOutput.toByteArray();
        }

        System.out.println("--- cloning ---");
        JumpCounter otherCounter = null;
        try(ByteArrayInputStream bytesInput = new ByteArrayInputStream(bytes);
            ObjectInputStream objectInput = new ObjectInputStream(bytesInput)) {
            otherCounter = (JumpCounter)objectInput.readObject();
        }

        System.out.println(counter.jump(1));
        System.out.println(otherCounter.jump(2));
    }
}
```

Listing 2.8: Mehrfache Deserialisierung eines Objekts.

Die Ausgabe des Programms zeigt, dass die zwei `JumpCounter`, Original und Clone, unabhängige Objekte sind, obwohl nur *ein* Konstruktoraufruf aufgezeichnet

wird:

```
$ java CounterClones
Counter()
JumpCounter()
--- cloning ---
JumpCounter@6
JumpCounter@7
```

Auf diesem Weg können beliebig viele Clones erzeugt werden. Der folgende Code ersetzt den zweiten Teil der `main`-Methode des vorhergehenden Beispiels. Es deserialisiert das Byte-Array in einer Schleife in viele unabhängige Clones und zählt diese unterschiedlich weiter:

```
import java.io.*;
import java.util.*;

public class CounterClonesList {
    public static void main(String... args) throws IOException, ClassNotFoundException {
        byte[] b = null;
        JumpCounter counter = new JumpCounter().jump(3).jump(2);
        try(ByteArrayOutputStream bytesOutput = new ByteArrayOutputStream();
            ObjectOutputStream objectOutput = new ObjectOutputStream(bytesOutput)) {
            objectOutput.writeObject(counter);
            objectOutput.flush();
            b = bytesOutput.toByteArray();
        }

        System.out.println("--- cloning ---");
        List<JumpCounter> list = new ArrayList<>();
        for(int i = 0; i < 5; i++)      // Liste mit Clones füllen
            try(ByteArrayInputStream input = new ByteArrayInputStream(b);
                ObjectInputStream objectInput = new ObjectInputStream(input)) {
                list.add((JumpCounter)objectInput.readObject());
            }

        System.out.println(list);       // Liste ausgeben
        for(int i = 0; i < 5; i++)      // alle unterschiedlich hochzählen
            list.get(i).jump(i);
        System.out.println(list);       // Liste noch einmal ausgeben
    }
}
```

Listing 2.9: Vielfache Deserialisierung eines Objekts.

Auch hier zeigt sich, dass unabhängige Objekte ohne Konstruktoraufrufe entstehen:

```
$ java CounterClones
```

```
Counter()
JumpCounter()
--- cloning ---
[JumpCounter@5, JumpCounter@5, JumpCounter@5, JumpCounter@5, JumpCounter@5]
[JumpCounter@5, JumpCounter@6, JumpCounter@7, JumpCounter@8, JumpCounter@9]
```

Alternatives Interface `Cloneable`

Java bietet mit der Methode `clone` des Interface `Cloneable` bereits einen Mechanismus an, um Objekte ohne Konstruktoraufrufe zu duplizieren. Allerdings hat eine korrekte Implementierung des Interface `Cloneable` eine ganze Reihe von eigenen Tücken und Einschränkungen.

2.2.7 Caching

Mehrfache Serialisierung eines Objekts

Ein Objekt wird nur einmal serialisiert, auch wenn es mehrfach referenziert wird. Bei Objektgraphen ist das notwendig, weil sonst die Struktur nicht originalgetreu restauriert werden könnte. Diese Regelung gilt allerdings für einen kompletten Object-Stream. Das bedeutet, dass ein Objekt nur *einmal* serialisiert wird, auch wenn es später noch einmal ausgegeben wird, vielleicht als Teil einer anderen Objektstruktur. Änderungen von Objekten *nach* der ersten Serialisierung schlagen sich damit nicht im Object-Stream nieder, wie im folgenden Beispiel:

```
import java.io.*;

public class ChangedCounterIO {
    public static void main(String... args) throws IOException, ClassNotFoundException {
        if(args.length == 1)
            try(OutputStream output = new FileOutputStream(args[0]);
                ObjectOutputStream objectOutput = new ObjectOutputStream(output)) {
                Counter counter = new Counter();
                objectOutput.writeObject(counter);
                counter.step();
                objectOutput.writeObject(counter);
            }
        else
            try(InputStream input = new FileInputStream(args[0]);
                ObjectInputStream objectInput = new ObjectInputStream(input)) {
                System.out.println(objectInput.readObject());
                System.out.println(objectInput.readObject());
            }
    }
}
```

Listing 2.10: Serialisierung eines Objekts vor und nach einer Änderung.

Rückwärtsverweise im Object-Stream

Die zweite Serialisierung des*selben* Objekts mündet nur in einen Rückverweis. Die Ausgabe zeigt deshalb zweimal den ursprünglichen Wert 0:

```
$ java ChangedCounterIO objects.ser
$ java ChangedCounterIO objects.ser read
Counter@0
Counter@0           // nicht: Counter@1
```

Die Methode `writeUnshared` der Klasse `ObjectOutputStream` serialisiert ein Objekt immer neu und erzeugt nie einen Rückwärtsverweis.

2.2.8 Nicht serialisierte Variablen

Keine Serialisierung von Klassenvariablen

Klassenvariablen sind von der Serialisierung ausgenommen, weil sie nicht einzelnen Objekten zugeordnet sind. Sie „gehören“ einer ganzen Klasse und werden als Teil des Codes behandelt, ebenso wie die Methoden einer Klasse. Das führt allerdings nicht immer zum gewünschten Verhalten.

Die folgende Klasse `StampedCounter` erweitert `Counter` um eindeutige „Seriennummern“. Die Klassenvariable `nextId` enthält die nächste freie Seriennummer, die Objektvariable `id` die tatsächliche Seriennummer eines Zählers. `toString` ergänzt die Textdarstellung um die Seriennummer.

```
public class StampedCounter extends Counter {
    private static int nextId = 1;

    private final int id = nextId++;

    public StampedCounter() {
        System.out.println("StampedCounter()");
    }

    public String toString() {
        return super.toString() + "#" + id;
    }
}
```

Listing 2.11: Serialisierung mit Klassenvariablen.

Das Programm `StampedCounterIO` serialisiert zwei `StampedCounter`. Beim Aufruf mit zwei Argumenten werden die beiden Zähler wieder deserialisiert und zusätzlich wird ein neuer dritter `StampedCounter` erzeugt.

```
import java.io.*;

public class StampedCounterIO {
    public static void main(String... args) throws IOException, ClassNotFoundException {
        if(args.length == 1)
```

```
            try(OutputStream output = new FileOutputStream(args[0]);
                ObjectOutputStream objectOutput = new ObjectOutputStream(output)) {
                objectOutput.writeObject(new StampedCounter());
                objectOutput.writeObject(new StampedCounter().step());
            }
        else
            try(InputStream input = new FileInputStream(args[0]);
                ObjectInputStream objectInput = new ObjectInputStream(input)) {
                System.out.println(objectInput.readObject());
                System.out.println(objectInput.readObject());
                System.out.println(new StampedCounter());
            }
    }
}
```

Listing 2.12: Schreiben und Lesen mehrerer Objekte.

Hier zeigt sich das Problem: Statt der nächsten Seriennummer (3) erhält der neue `StampedCounter` wieder die Seriennummer 1, die allerdings schon früher vergeben wurde. Die „eindeutigen“ Seriennummern sind nicht eindeutig!

```
$ java StampedCounterIO objects.ser
Counter()
StampedCounter()
Counter()
StampedCounter()
$ java StampedCounterIO objects.ser read
StampedCounter@0#1
StampedCounter@1#2
Counter()
StampedCounter()
StampedCounter@0#1      // nicht: StampedCounter@0#3
```

Anwendungen selbst verantwortlich für Klassenvariablen

Eine Anwendung muss selbst Vorkehrungen treffen, um Klassenvariablen zu sichern. Eine Möglichkeit dazu wird auf Seite 144 vorgestellt.

Explizite Ausnahme von der Serialisierung

Auf der anderen Seite ist auch nicht immer sinnvoll, dass *jede* Objektvariable von der Serialisierung erfasst wird. Der Modifier `transient` schließt eine Objektvariable gezielt von der Serialisierung aus.

Für die nächste Klasse `CachedStampedCounter` wird um des Beispiels willen angenommen, dass die String-Darstellung nur mit großem Aufwand erzeugt werden könnte. Eine wiederholte Berechnung der gleichen String-Darstellung soll möglichst vermieden werden, deshalb merkt sich `toString` das Ergebnis beim ersten Aufruf in der Objektvariablen `cached` und liefert es bei weiteren Aufrufen ohne Neuberechnung zurück. Zusätzlich protokolliert `toString`, ob ein gespeichertes Ergebnis verwendet werden konnte (Ausgabe von „hit“) oder ob eine Neuberechnung nötig war (Ausgabe von „miss“). `step` löscht den „Cache“, weil die bisher

gemerkte String-Darstellung dann nicht mehr stimmt und beim nächsten Zugriff neu berechnet werden muss.

```
public class CachedStampedCounter extends StampedCounter {
    private String cached = null;

    public Counter step() {
        cached = null;
        return super.step();
    }

    public String toString() {
        if(cached == null) {
            cached = '*' + super.toString();
            System.out.println("miss");
        }
        else
            System.out.println("hit");
        return cached;
    }
}
```

Listing 2.13: Klasse mit einer temporären Objektvariablen.

Das Programm `CachedStampedCounterIO` erzeugt einen `CachedStampedCounter`, gibt ihn zweimal aus und serialisiert ihn. Nach dem Deserialisieren wird er erneut zweimal ausgegeben.

```
import java.io.*;
import static java.lang.System.*;

public class CachedStampedCounterIO {
    public static void main(String... args) throws IOException, ClassNotFoundException {
        if(args.length == 1)
            try(OutputStream output = new FileOutputStream(args[0]);
                ObjectOutputStream objectOutput = new ObjectOutputStream(output)) {
                Counter counter = new CachedStampedCounter();
                out.println(counter);
                out.println(counter);
                objectOutput.writeObject(counter);
            }
        else
            try(InputStream input = new FileInputStream(args[0]);
                ObjectInputStream objectInput = new ObjectInputStream(input)) {
                Counter counter = (Counter)objectInput.readObject();
                out.println(counter);
                out.println(counter);
            }
    }
}
```

Listing 2.14: Ein- und Ausgabe komplexer Objekte via Object-Streams.

Wie erwartet wird mit dem ganzen Objekt auch die Variable `cached` serialisiert. Nach dem Restaurieren wird weiterhin ohne Neuberechnung das gemerkte Ergebnis benutzt.

```
$ java CachedStampedCounterIO objects.ser
Counter()
StampedCounter()
miss
*CachedStampedCounter@0#1
hit
*CachedStampedCounter@0#1

$ java CachedStampedCounterIO objects.ser read
hit
*CachedStampedCounter@0#1
hit
*CachedStampedCounter@0#1
```

Wirkung von `transient`

Unterstellt man jetzt noch, dass der Cache sehr umfangreich ist, dann sollte er besser von der Serialisierung ausgenommen werden, um die Object-Streams kompakt zu halten und nicht mit redundanten Informationen aufzublähen. Das erreicht man mit dem Modifier `transient`:

```
private  transient String cached = null;
```

Ein neuer Lauf des Programms `CachedStampedCounterIO` (Listing 2.14) zeigt den Unterschied. Der Cache fehlt jetzt im Object-Stream, nach dem Deserialisieren ist er leer. `toString` muss die String-Darstellung beim ersten Aufruf neu berechnen:

```
$ java CachedStampedCounterIO objects.ser
Counter()
StampedCounter()
miss
*CachedStampedCounter@0#1
hit
*CachedStampedCounter@0#1

$ java CachedStampedCounterIO objects.ser read
miss                    // statt: hit
*CachedStampedCounter@0#1
hit
*CachedStampedCounter@0#1
```

Bei jeder Klassendefinition ist im Einzelfall zu entscheiden, ob und welche Variablen `transient` sein sollen.

2.2.9 Serialisierungsversionen

Kennnummern (UIDs) für Codeversionen

Object-Streams enthalten außer den serialisierten Objekten selbst noch zusätzliche Codes, die die beim Serialisieren verwendeten Klassendefinitionen eindeutig kennzeichnen. Diese **Serial Version Unique Identifier** (im Rest des Kapitels kurz als UIDs bezeichnet) berechnen sich aus dem Namen der Klasse und allen Elementen samt ihren Modifiern. Das Kommandozeilenwerkzeug `serialver` zeigt die UID einer Klasse:[10]

```
$ serialver Counter
Counter: static final long serialVersionUID = 734632963106752125L;
```

Keine Deserialisierung bei Abweichung der UID

Die Deserialisierung vergleicht die UID im Object-Stream mit der UID der vorliegenden Klassendefinition. Sie bricht bei einer Abweichung mit einer `InvalidClassException` ab.

Beispielsweise kann man mit dem Programm `CounterIO` (Listing 2.3) eine Datei mit serialisierten Zählern schreiben. Ändert man nach dem Serialisieren die Definition von `Counter`, dann lässt sich die Datei nicht mehr einlesen. Dazu reicht es zum Beispiel aus, die Sichtbarkeit einer Methode zu ändern.[11]

```
$ java CounterIO objects.ser read
InvalidClassException: Counter; local class incompatible:
stream classdesc serialVersionUID = 734632963106752125,
local class serialVersionUID = -6382845449450593138
```

Java verhält sich hier defensiv. Die Verträglichkeit von Codeversionen mit serialisierten Objekten kann man aber selbst in die Hand nehmen. Die Klassenvariable[12]

```
private static final long serialVersionUID
```

Explizite Vergabe von UIDs

legt die UID explizit fest und ersetzt die automatische Berechnung aus den Einzelteilen der Klassendefinition. In der Klasse `Counter` kann man beispielsweise einen bestimmten Wert festsetzen:[13]

[10] `serialver` ist Teil des JDK.

[11] Ohne Auswirkung bleiben dagegen beispielsweise Änderungen innerhalb von Methodenrümpfen, Änderung der Definitionsreihenfolge von Methoden oder Änderungen von Parameternamen.

[12] Die Sichtbarkeit ist formal nicht relevant, sollte aber `private` sein. Eine UID gehört zum inneren Aufbau einer Klasse und ist für andere Klassen ohne Wert.

[13] Der konkrete Zahlenwert der UID ist irrelevant. Eine Zeitmarke hat sich bewährt.

```
import java.io.*;

public class Counter implements Serializable {
    private static final long serialVersionUID = 20110912;
    // ...
}
```

Listing 2.15: Serialisierbare Klasse mit expliziter Serialisierungsversion.

Kommandozeilenwerkzeug `serialver`

Das Werkzeug `serialver` berichtet den expliziten Wert:

```
$ serialver Counter
Counter: static final long serialVersionUID = 20110912;
```

2.2.10 Eingriff in die Serialisierung

Benutzerdefinierte Serialisierung

Abgesehen von `transient` lässt sich die automatische „Standard-Serialisierung“ zunächst nicht beeinflussen. Das ist in vielen Fällen ausreichend, nicht aber in allen. Die Standard-Serialisierung übergeht zum Beispiel die Klassenvariable `nextId` der Klasse `StampedCounter` (Listing 2.11), obwohl sie gebraucht wird.

Private Hilfsmethoden für Sonderfälle

In solchen Fällen kann mit den Methoden

```
private void  writeObject(ObjectOutputStream out)
private void  readObject(ObjectInputStream in)
```

in die Serialisierung eingegriffen werden. Diese Methoden werden, falls sie definiert sind, bei der Serialisierung aufgerufen.[14] Sie erhalten jeweils einen offenen `ObjectStream` als Argument und sind dafür verantwortlich, das eigene Objekt auszugeben beziehungsweise einzulesen. Zur Vereinfachung können sie dabei auf die Methoden `defaultWriteObject` und `defaultReadObject` der Klassen `ObjectOutputStream` und `ObjectInputStream` zurückgreifen, die die Standard-Serialisierung abwickeln.[15]

Als Beispiel wird die Klasse `StampedCounter` (Listing 2.11) um eine explizite Sicherung der Klassenvariablen erweitert. Beim Schreiben und Lesen der Objekte wird zuerst die Standard-Serialisierung angestoßen, die sich um die Objektvariablen und die Basisklasse kümmert. Anschließend wird noch der Wert der Klassenvariablen `nextId` nachgeschoben:

[14] Die Methoden sind `private` und gehören zu keinem Interface. Sie werden über Reflection gefunden und aufgerufen (Kapitel 8).

[15] `defaultWriteObject` und `defaultReadObject` dürfen *nur von* `writeOject` und `readObject` aufgerufen werden.

```
import java.io.*;

public class CustomStampedCounter extends Counter {
    // ... wie StampedCounter

    private void writeObject(ObjectOutputStream objectOutput) throws IOException {
        objectOutput.defaultWriteObject();
        objectOutput.writeInt(nextId);
    }

    private void readObject(ObjectInputStream objectInput) throws IOException, ClassNotFo
        objectInput.defaultReadObject();
        nextId = objectInput.readInt();
    }
}
```

Listing 2.16: Klasse mit eigenen Methoden zur Serialisierung.

Die beiden Methoden müssen sich natürlich einig sein. Das Basisklassenobjekt (`Counter`) wird unabhängig und automatisch serialisiert. `writeObject` und `readObject` haben darauf keinen Einfluss.

Anders als vorher (Seite 140) überlebt die Klassenvariable jetzt die Serialisierung. Das Programm `CustomStampedCounterIO` ist identisch mit `StampedCounterIO` (Listing 2.12), verwendet aber `CustomStampedCounter` (Listing 2.16) statt `StampedCounter` (Listing 2.11). Das nach der Deserialisierung als Nächstes erzeugte Objekt erhält jetzt die nächste freie Seriennummer, 3.[16]

```
$ java CustomStampedCounterIO objects.ser
Counter()
CustomStampedCounter()
Counter()
CustomStampedCounter()
$ java CustomStampedCounterIO objects.ser read
CustomStampedCounter@0#1
CustomStampedCounter@1#2
Counter()
CustomStampedCounter()
CustomStampedCounter@0#3        // vorher: ... #0
```

Die beiden Methoden `writeObject` und `readObject` müssen nicht unbedingt paarweise definiert werden, sondern sind auch einzeln sinnvoll verwendbar. Im folgenden Beispiel werden die Seriennummern beim Einlesen von `readObject` kurzerhand neu zugewiesen. Zum einen erübrigt sich damit `writeObject`, zum anderen

[16] Das Problem der eindeutigen Seriennummern ist damit immer noch nicht endgültig gelöst. Dieser simple Ansatz funktioniert nur dann, wenn neue Objekte *nach* der Deserialisierung geschaffen werden. Bei einer anderen Reihenfolge können wieder mehrdeutige Seriennummern vergeben werden.

braucht die Seriennummer überhaupt nicht mehr serialisiert zu werden und kann als `transient` markiert werden.

```
import java.io.*;
import static java.lang.System.*;

public class SerialStampedCounter extends Counter {
    private transient int id = nextId++;
    // ... sonst wie StampedCounter, ohne writeObject

    private void readObject(ObjectInputStream objectInput) throws IOException, ClassNotFo
        objectInput.defaultReadObject();
        id = nextId++;
    }

}
```

Listing 2.17: Neu-Initialisierung einer transienten Variablen bei der Deserialisierung.

Jetzt werden zwar zuverlässig eindeutige Seriennummern vergeben, allerdings bleiben sie bei der Serialisierung nicht erhalten.

Interface `Externalizable` für genaue Kontrolle

Weitergehende Kontrolle über die Serialisierung erlaubt das von `Serializable` abgeleitete Interface `Externalizable`. Dieses Interface deklariert die zwei Methoden `writeExternal` und `readExternal`, denen die Serialisierung komplett übertragen wird. Anders als `writeObject` und `readObject` sind sie auch für die Basisklasse verantwortlich.

2.3 JavaBeans

Alternative zu Object-Streams

Ursprünglich wurden JavaBeans (im Rest des Kapitels einfach „Beans“)[17] geschaffen, um eine komfortable Entwicklung grafischer Benutzeroberflächen (*Graphical User Interface*, GUI) zu ermöglichen. Die Komponenten einer GUI, also Dialogelemente wie Buttons, Eingabefelder, Beschriftungen, Pop-up-Boxen und dergleichen, und deren Anordnung sollten einfach erzeugt, bearbeitet, gesichert und wieder geladen werden können. Unter anderen Eigenschaften von Beans wird hier die Umsetzung der Serialisierung beleuchtet, die unter bestimmten Voraussetzungen eine interessante Alternative zu Object-Streams bietet. Beans erweisen sich dabei auch abseits von GUIs als nützliches Konzept.

[17] Es gibt auch „Enterprise JavaBeans“. Diese spielen in der Java Enterprise Edition (JEE) eine wichtige Rolle, um die es hier aber nicht geht.

2.3.1 Properties

Sicherung von Eigenschaften

Anders als bei Object-Streams wird bei Beans nicht der innere Aufbau von Objekten in einer Bytefolge festgehalten, sondern Code zur Rekonstruktion von Objekten aufgezeichnet. Bei der Deserialisierung wird dieser Code ausgeführt und reproduziert wieder die ursprünglich serialisierten Objekte. Die Innenstruktur von Objekten spielt keine Rolle mehr, weil nur noch Methoden zum Aufbau gebraucht werden.

Formale Anforderungen an die Klassendefinition

Die Eigenschaften von Objekten werden im Zusammenhang mit Beans als **Properties** bezeichnet. Beans stellen einige entscheidende Anforderungen: Es muss

- für jede Property eine Methode zum Abfragen (Getter),
- für jede Property eine Methode zum Zuweisen (Setter) und
- einen Default-Konstruktor

geben. Klassen, die diesen Anforderungen genügen, können als Beans serialisiert werden.[18]

Getter und Setter müssen den bekannten Namenskonventionen folgen. Zu einer Property

```
type name
```

lauten die Signaturen von Getter und Setter verbindlich:[19]

```
public type getName()
public void setName(type)
```

Der Getter einer `boolean`-Property kann wahlweise auch definiert sein als

```
public boolean isName()
```

Neben den einfachen Properties gibt es auch indexierte Properties, die in Arrays übergeben werden.[20] Zu einer indexierten Property sind die folgenden vier Methoden zu definieren:

[18] Später zeigt sich, dass auf diese Forderungen bei geeigneten Vorbereitungen verzichtet werden kann.

[19] Leider schreiben die Beans-Konventionen den Ergebnistyp `void` für Setter vor und verhindern damit die Umsetzung eines *fluent interface*. In diesem Idiom liefern Setter das eigene Objekt zurück und erlauben dadurch elegante Kettenaufrufe.

[20] Das heißt nicht, dass die Bean-Klasse intern auch Arrays verwenden muss. Hier geht es nur um Methodenschnittstellen.

```
public type getName(int)
public void setName(int, type)
public type[] getName()
public void setName(type[])
```

Die ersten beiden liefern beziehungsweise ersetzen einen einzelnen Wert am gegebenen Index. Die letzten beiden übermitteln alle Werte auf einmal als Array.

Beispiel einer JavaBean-Klasse

Die folgende Klasse `PlayerBean` repräsentiert einen Spieler in einem hypothetischen Spiel. Die Properties eines Spielers sind

- der Name (`String name`) und
- der Punktestand (`int score`).

Die Klassendefinition folgt den Beans-Konventionen: [21]

```
public class PlayerBean {
    private String name = null;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    private int score = 0;

    public int getScore() {
        return score;
    }

    public void setScore(int score) {
        this.score = score;
    }
}
```

Listing 2.18: Bean-Klasse mit Gettern und Settern.

JavaBean mit indexierter Property

Die weitere Bean-Klasse `PlayersBean` kapselt mehrere Spieler als indexierte Property.[22] Obwohl gemäß Konventionen Arrays von Spielern übergeben werden, arbeitet die Klasse intern mit einer Liste.

[21] Der Code einer solchen Klassendefinition lässt sich oft auf Knopfdruck generieren.

[22] Die Benennung der Methoden passt nicht recht: `setPlayers` fügt einen oder alle Spieler ein. Der Plural im Namen passt für den zweiten Fall, aber nicht für den ersten. Mit dem Singular wäre es umgekehrt, aber auch nicht besser. Wegen der Namenskonventionen muss das hingenommen werden.

```
import java.util.*;

public class PlayersBean {
    private List<PlayerBean> players = new ArrayList<>();

    public PlayerBean[] getPlayers() {
        return players.toArray(new PlayerBean[0]);
    }

    public void setPlayers(PlayerBean[] players) {
        this.players = Arrays.asList(players);
    }

    public PlayerBean getPlayers(int index) {
        return players.get(index);
    }

    public void setPlayers(int index, PlayerBean player) {
        players.set(index, player);
    }
}
```

Listing 2.19: Bean-Klasse mit einer indexierten Property.

2.3.2 `XMLEncoder` und `XMLDecoder`

Bibliotheksmethoden zum Serialisieren und Deserialisieren

Die Klassen `XMLEncoder` und `XMLDecoder` im Package `java.beans` definieren Methoden zum Serialisieren und Deserialisieren von Beans:

```
void  writeObject(Object x)
Object  readObject()
```

Die Konstruktoren der Klassen akzeptieren einen Bytestrom als Ziel beziehungsweise Quelle. `XMLEncoder` und `XMLDecoder` müssen mit Aufrufen von `close` geschlossen werden.[23] Das Programm `PlayersBeanIO` serialisiert oder deserialisiert eine `PlayersBean`-Bean, abhängig von der Anzahl der Kommandozeilenargumente:

```
import java.beans.*;
import java.io.*;

public class PlayersBeanIO {
    public static void main(String... args) throws IOException {
        PlayersBean players;
        if(args.length == 1)
            try(OutputStream output = new FileOutputStream(args[0]);
                XMLEncoder encoder = new XMLEncoder(output)) {
                PlayerBean max = new PlayerBean();
```

[23] Sie implementieren das Interface `AutoCloseable`.

```
                max.setName("Max");
                max.setScore(5);

                PlayerBean moritz = new PlayerBean();
                moritz.setName("Moritz");
                moritz.setScore(3);

                players = new PlayersBean();
                players.setPlayers(new PlayerBean[] {max, max});

                encoder.writeObject(players);
            }
        else
            try(InputStream input = new FileInputStream(args[0]);
                XMLDecoder decoder = new XMLDecoder(input)) {
                players = (PlayersBean)decoder.readObject();
            }
        System.out.println(players);
    }
}
```

Listing 2.20: Schreiben und Lesen von JavaBeans als XML.

XML-Dokument als Serialisierungsformat

Die Aufrufe von `writeObject` und `readObject` unterscheiden sich nicht von denen der Object-Streams. Das Format der serialisierten Objekte ist allerdings vollkommen anders. Die XML-Coder verwenden ein lesbares, portables XML-Dokument (Kapitel 3), im Gegensatz zum binären Format der Object-Streams. Das folgende Beispiel zeigt ein solches XML-Dokument:[24]

```
$ java PlayersBeanIO players.xml
[Max/5, Moritz/3]
$ java PlayersBeanIO players.xml read
[Max/5, Moritz/3]
$ cat players.xml
<?xml version="1.0" encoding="utf-8"?>
<java version="1.7.0" class="java.beans.XMLDecoder">
    <object class="PlayersBean">
        <void property="players">
            <array class="PlayerBean" length="2">
                <void index="0">
                    <object class="PlayerBean">
                        <void property="name">
                            <string>Max</string>
                        </void>
                        <void property="score">
                            <int>5</int>
                        </void>
                    </object>
                </void>
                <void index="1">
```

[24] Die ersten beiden Aufrufe benutzen `toString`-Methoden der Klassen `PlayerBean` und `PlayersBean`, die im oben abgedruckten Code weggelassen sind.

```
                    <object class="PlayerBean">
                        <void property="name">
                            <string>Moritz</string>
                        </void>
                        <void property="score">
                            <int>3</int>
                        </void>
                    </object>
                </void>
            </array>
        </void>
    </object>
</java>
```

Dieses Dokument enthält keine Information über den Aufbau der serialisierten Objekte, sondern nur deren Properties und Werte.

2.3.3 Speicherformat XML

Merkmale von XML als Speicherformat

XML-Dokumente weisen gegenüber einem binären Serialisierungsformat einige Vorteile auf:

Standardisiert
: XML kann von praktisch allen Systemen und allen Programmiersprachen verarbeitet werden. Es gibt darüber hinaus ein großes Angebot an Werkzeugen zum Umgang mit XML-Dokumenten.

Lesbar
: Als Textformat kann es mit einem Editor oder anderen textverarbeitenden Werkzeugen untersucht werden, notfalls sogar durch Augenschein.

Editierbar
: XML kann mit einem Texteditor korrigiert, erweitert und auf anderen Wegen angepasst werden. Grundsätzlich könnte es auch mit Java-fremden Systemen oder gar von Hand neu erstellt werden.

Langlebig
: Texte eignen sich zur Archivierung.[25]

Versionsunabhängig
: Die Klassendefinitionen der serialisierten Objekte können unbegrenzt verändert werden, solange nur die Getter und Setter erhalten bleiben.

[25] Bis heute haben sich Ausdrucke als langlebigster Datenträger erwiesen.

Dem stehen auch Nachteile gegenüber:

Volumen
: XML ist als Textformat voluminöser als ein binäres Format.

Effizienz
: Schreiben und Lesen von XML ist verhältnismäßig langsam.

Manipulation mit einem Editor

Zum Beispiel kann an die Spielerliste mit einem Texteditor ein weiteres Element angefügt werden. Die Länge des Arrays in der Zeile `<array...>` muss dabei ebenfalls angepasst werden.

```
<?xml version="1.0" encoding="utf-8"?>
<java version="1.7.0" class="java.beans.XMLDecoder">
    <object class="PlayersBean">
        <void property="players">
            <array class="PlayerBean" length="3">
                <void index="0">
                    ...
                </void>
                <void index="1">
                    ...
                </void>
                <void index="2">
                    <object class="PlayerBean">
                        <void property="name">
                            <string>Fromme Helene</string>
                        </void>
                        <void property="score">
                            <int>4</int>
                        </void>
                    </object>
                </void>
            </array>
        </void>
    </object>
</java>
```

Das veränderte Dokument lässt sich fehlerfrei deserialisieren:

```
$ java PlayersBeanIO players.xml read
[Max/5, Moritz/3, Fromme Helene/4]
```

2.3.4 Inkonsistenzen

Umgang mit Fehlern bei der Deserialisierung

Die Manipulation serialisierter XML-Dokumente ist möglich, aber sicher nur ein

letzter Ausweg. Die Einhaltung von Syntax und Semantik liegen in diesem Fall alleine in der Hand des Editors, ob Mensch oder Maschine. Der `XMLDecoder` kommt mit fehlerhaften Dokumenten bis zu einem gewissen Grad zurecht. Im Zuge der Deserialisierung werden zwar Exceptions ausgelöst, diese werden aber vom `XMLDecoder` selbst abgefangen und die Deserialisierung wird fortgesetzt. Im folgenden Beispiel ist in die serialisierte Spielerliste eine *nicht existierende* Property „simsalabim“ eingefügt:

```
... <void index="2">
    <object class="PlayerBean">
        <void property="name">
            <string>Max</string>
        </void>
        <void property="score">
            <int>5</int>
        </void>
        <void property="simsalabim">   <!-- nicht definiert ->
            <double>3.14159263</double>
        </void>
    </object>
</void> ...
```

Bei dem Versuch diese Datei einzulesen wird die überzählige Property erkannt und ignoriert:

```
$ java PlayersBeanIO players.xml read
java.lang.NoSuchMethodException: <unbound>=PlayerBean.setSimsalabim(Double);
Continuing ...
[Max/5, Moritz/3]
```

Fehlende Property-Angaben führen zu Defaultwerten der deserialisierten Objekte. Es ist offen, wie weit semantisch fehlerhafte Dateien noch brauchbare Objekte liefern.

2.3.5 Objektgraphen

Rückreferenzen bei mehrfach serialisierten Objekten

Beans enthalten Properties mit primitiven Werten oder mit Referenzen auf weitere Beans. Sie bilden also einen Objektgraphen, der bei der Serialisierung durchlaufen und ausgegeben wird.

Mehrfach referenzierte Objekte werden nur beim ersten Besuch serialisiert und später nur noch referenziert. Auch `null`, Selbstreferenzen und Zyklen werden erkannt und korrekt verarbeitet. Als Beispiel dient ein `PlayersBean`-Objekt mit zweimal dem*selben* `PlayerBean`-Element:

```
players.setPlayers(new PlayerBean[] {max, max});
```

In der serialisierten Form findet sich nur ein Objekt und eine Referenz darauf, die mit den XML-Attributen `id` und `idref` hergestellt wird (siehe Seite 187):

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.7.0" class="java.beans.XMLDecoder">
    <object class="PlayersBean">
        <void property="players">
            <array class="PlayerBean" length="2">
                <void index="0">
                    <object class="PlayerBean" id="PlayerBean0">
                        <void property="name">
                            <string>Max</string>
                        </void>
                        <void property="score">
                            <int>5</int>
                        </void>
                    </object>
                </void>
                <void index="1">
                    <object idref="PlayerBean0" />
                </void>
            </array>
        </void>
    </object>
</java>
```

2.3.6 Serialisierer-Objekte

Arbeitsweise der Serialisierung

Der `XMLEncoder` arbeitet mit einer Tabelle, die jedem Objekttyp einen „Serialisierer“ zuordnet. Ein Serialisierer ist zuständig für die Serialisierung von Werten des betreffenden Typs. So gibt es beispielsweise für jeden primitiven Typ einen passenden Serialisierer, ebenso für Arrays und Strings.

Alle anderen Referenztypen werden dem „Default-Serialisierer“ übergeben, der in der Klasse `DefaultPersistenceDelegate` definiert ist. Der Default-Serialisierer arbeitet recht einfach: Er ruft nacheinander alle Getter des Objekts auf und serialisiert deren Rückgabewerte rekursiv nach dem gleichen Verfahren. Dabei berücksichtigt er mehrfach referenzierte Objekte, wie oben beschrieben.

Anpassung der Serialisierung

Die Methode `setPersistenceDelegate` von `XMLEncoder` ändert die Tabelle. Sie ordnet einem bestimmten Typen einen neuen Serialisierer zu. Damit lassen sich beispielsweise Klassen mit unveränderlichen Properties serialisieren. Die folgende Bean `Player` entspricht der weiter vorne definerten Klasse `PlayerBean` (Listing 2.18), allerdings jetzt mit einem unveränderlichen Namen.[26] Die entsprechende Objekt-

[26] Für Spieler passt das ohnedies besser. Sie sollten wohl kaum mitten im Spiel den Namen ändern.

variable `name` ist hier `final` und wird im Konstruktor initialisiert. Der Setter fällt weg.

```
public class Player {
    private final String name;

    public Player(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    private int score = 0;

    public int getScore() {
        return score;
    }

    public void setScore(int score) {
        this.score = score;
    }
}
```

Listing 2.21: Bean-Klasse mit veränderlichen und unveränderlichen Properties.

Unveränderliche Properties

Ein überladener `DefaultPersistenceDelegate`-Konstruktor akzeptiert als Argument ein String-Array mit den Namen der unveränderlichen Properties. Das folgende Programmfragment ersetzt im `encoder` den Default-Serialisierer des Typs `Player` durch einen neuen Serialisierer, der die Property „name“ als unveränderlich behandelt:[27]

```
try(OutputStream output = new FileOutputStream(args[0]);
    XMLEncoder encoder = new XMLEncoder(output)) {
    encoder.setPersistenceDelegate(Player.class,
                                   new DefaultPersistenceDelegate(new String[] {
                                           "name"
                                   }));
```

Listing 2.22: Ersatz des Default-Serialisierers durch einen neuen Serialisierer.

An der Deserialisierung ist keine Änderung nötig, wie im nächsten Abschnitt begründet wird.

[27] Die Situation ist so häufig, dass die Klasse `DefaultPersistenceDelegate` diese einfache Lösung dafür bereitstellt.

2.3.7 Neue Serialisierer

Einbau eigener Serialisierer

`XMLEncoder` arbeitet auch mit neuen, selbst definierten Serialisierern. Zum Beispiel sind in der weiter vorne definierten Klasse `PlayersBean` (Listing 2.19) vier Methoden erforderlich, um die indexierte Property `players` getreu Konvention umzusetzen. Diese Bean-Klasse weist Eigenschaften auf, die zum Teil etwas störend sind:

- Die intern verwendete Liste muss in Settern und Gettern in Arrays konvertiert werden.
- Der Array-Setter übernimmt eine Referenz in das Objekt, die der Aufrufer bereitstellt. Eine solche „externe“ Referenz aus fremder Quelle widerspricht der Datenkapselung, weil der Aufrufer im Nachhinein das betreffende Objekt nach Belieben manipulieren kann.
- Die `void`-Ergebnistypen verhindern ein „Fluent Interface“.
- Der volle Satz aller vier Methoden ist etwas ausladend und wird vielleicht nicht gebraucht.

Wegfall der Beans-Konventionen

Die folgende Klasse `Players` räumt mit diesen Problemen auf und bietet eine kompaktere und sicherere Schnittstelle. Sie folgt allerdings überhaupt nicht mehr den eingangs vorgestellten Beans-Konventionen.

```
import java.util.*;

public class Players {
    private final List<Player> players = new ArrayList<>();

    public List<Player> getPlayers() {
        return Collections.unmodifiableList(players);
    }

    public Players addPlayer(Player player) {
        players.add(player);
        return this;
    }
}
```

Listing 2.23: Klasse mit einer Liste von Spielern.

Mit einem passenden Serialisierer kann auch diese Klasse verwendet werden. Der Serialisierer muss allerdings dafür sorgen, dass ein `Players`-Objekt bei der Deserialisierung durch `addPlayer`-Aufrufe mit deserialisierten `Player`-Objekten gefüllt wird. Dazu wird von `DefaultPersistenceDelegate` eine neue Klasse `PlayersPersistenceDelegate` abgeleitet, die die Methode `initialize` redefiniert. `initialize` legt fest, wie ein Objekt deserialisiert wird. Die Implementierung in der Basisklasse `DefaultPersistenceDelegate` sucht nach Methoden gemäß Beans-Konventionen, findet jetzt aber nichts mehr.

Der Aufruf `super.initialize` hat also nur noch einen Default-Konstruktoraufruf zur Folge. Anschließend wird in einer Schleife für jeden `Player` ein `Statement`-Objekt erzeugt, das einen Aufruf der Methode `addPlayer` repräsentiert.[28]

```
import java.beans.*;

public class PlayersPersistenceDelegate extends DefaultPersistenceDelegate {
    protected void initialize(Class type, Object old, Object noo, Encoder encoder) {
        super.initialize(type, old, noo, encoder);
        Players players = (Players)old;
        for(Player player: players.getPlayers()) {
            Statement statement = new Statement(old, "addPlayer", new Object[] {player});
            encoder.writeStatement(statement);
        }
    }
}
```

Listing 2.24: Serialisierer für eine Klasse mit einer Liste von Spielern.

Bei der Serialisierung wird ein `PlayersPersistenceDelegate` als Serialisierer für die Klasse `Players` registriert und dann automatisch vom `XMLEncoder` benutzt.

Registrieren des neuen Serialisierers

```
import java.beans.*;
import java.io.*;

public class PlayersIO {
    public static void main(String... args) throws IOException {
        Players players;
        if(args.length == 1)
            try(OutputStream output = new FileOutputStream(args[0]);
                XMLEncoder encoder = new XMLEncoder(output)) {
                encoder.setPersistenceDelegate(Player.class,
                                               new DefaultPersistenceDelegate(new String[
                                                       "name"
                                               }));

                Player max = new Player("Max");
                max.setScore(5);

                Player moritz = new Player("Moritz");
                moritz.setScore(3);

                players = new Players().addPlayer(max).addPlayer(moritz);
                encoder.writeObject(players);
            }
        else
            try(InputStream in = new FileInputStream(args[0]);
                XMLDecoder decoder = new XMLDecoder(in)) {
                players = (Players)decoder.readObject();
```

[28] Dieser nur für `Players` zuständige Serialisierer kann als Objektvariable in der Klasse `Players` selbst definiert werden. Die Implementierung vereinfacht sich dadurch noch ein wenig.

```
            }
        System.out.println(players);
    }
}
```

Listing 2.25: Ersatz des Default-Serialisierers durch einen neuen Serialisierer.

Wieder ist keine Änderung an der Deserialisierung nötig.

2.3.8 Deserialisierung

Arbeitsweise der Deserialisierung

Beim Deserialisieren liest ein `XMLDecoder` das XML-Dokument und baut Element für Element wieder die im Dokument beschriebenen Objekte zusammen. Die wesentlichen XML-Elemente sind dabei `object` und `void`.

- `object`-Elemente repräsentieren *Ausdrücke*, die eine Objektreferenz liefern.
- `void`-Elemente repräsentieren *Anweisungen*, die kein Ergebnis haben, sondern durch einen Seiteneffekt wirken. Sie richten sich an das Objekt, das dem übergeordneten `object`-Element entspricht.

XML-Elemente für Methodenaufrufe

In beiden Fällen legt das XML-Attribut `method` fest, welche Methode aufgerufen werden soll. Bei `object` kann das `method`-Attribut durch ein `class`-Attribut ersetzt werden und führt dann zu einem Konstruktoraufruf. Das Dokument

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.7.0" class="java.beans.XMLDecoder">
    <object class="Foo">
        <void method="run"/>
    </object>
</java>
```

entspricht dem Code

```
Foo f = new Foo();
f.run();
```

Das `void`-Attribut `property` ist lediglich eine Abkürzung für einen entsprechenden Setter-Aufruf.

Neben `object` gibt es noch Elemente für alle primitiven Typen sowie die Elemente `array` und `string`. Sie alle gehören zur Kategorie „Ausdrücke“ und produzieren beim Deserialisieren einen Wert des entsprechenden Typs.

Ausdrücke, die direkt Anweisungen oder anderen Ausdrücken untergeordnet sind, werden als Argumente für den betreffenden Methodenaufruf behandelt. Das Dokument

Geschachtelte Elemente für Argumente

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.7.0" class="java.beans.XMLDecoder">
    <object class="java.lang.System" field="out">
        <void method="println">
            <string>Hello, world!</string>
        </void>
    </object>
</java>
```

repräsentiert die Anweisung

```
System.out.println("Hello, world!");
```

Der weiter vorne entwickelte Serialisierer `PlayersPersistenceDelegate` (Listing 2.24) produziert beispielsweise das folgende XML-Dokument:

Serialisierungsdokument als Code

```
<?xml version="1.0" encoding="utf-8"?>
<java version="1.7.0" class="java.beans.XMLDecoder">
    <object class="Players">
        <void method="addPlayer">
            <object class="Player">
                <string>Max</string>
                <void property="score">
                    <int>5</int>
                </void>
            </object>
        </void>
        <void method="addPlayer">
            <object class="Player">
                <string>Moritz</string>
                <void property="score">
                    <int>3</int>
                </void>
            </object>
        </void>
    </object>
</java>
```

Dieses Dokument beschreibt etwa den folgenden Code:

```
Players pls = new Players();
Player p1 = new Player("Max");
```

```
p1.setScore(5);
pls.addPlayer(p1);
Player p2 = new Player("Moritz");
p2.setScore(3);
pls.addPlayer(p2);
```

Eine serialisierte Datei kann als Darstellung von Java-Code betrachtet werden, der bei Deserialisieren ausgeführt wird.

2.4 XStream

Externe Bibliothek mit flexiblen Eigenschaften

„XStream“ ist eine Bibliothek von Codehaus, einem Zusammenschluss von Entwicklern von Open-Source-Projekten. XStream ist im Internet frei verfügbar und steht unter einer liberalen BSD-Lizenz. Für die Zwecke dieses Abschnitts reicht es aus, die Jar-Datei `xstream-`*version*`.jar`[29] im Classpath zu installieren.

Mit XStream kann fast jede Klasse ohne weitere Anpassung serialisiert werden. Insbesondere muss kein Interface implementiert werden, wie etwa `Serializable` bei Object-Streams. Außerdem muss der Aufbau der Klassen keinen bestimmten Konventionen folgen wie bei Beans.

2.4.1 `toXML` und `fromXML`

XML als Serialisierungsformat

XStream verwendet XML-Elemente als Serialisierungsformat, genauso wie JavaBeans. Daraus ergeben sich die gleichen Vor- und Nachteile gegenüber dem binären Format der Object-Streams (Seite 151). Serialisieren und Deserialisieren werden im Wesentlichen über die Methoden

```
Object  fromXML(InputStream source)


void  toXML(Object x, OutputStream destination)
```

der Klasse `XStream` abgewickelt.

XML-Parser als Hilfsmittel

Zum eigentlichen Lesen und Schreiben des XML-Elements stützt sich XStream auf einen XML-Parser, der nicht in XStream enthalten ist. Es gibt verschiedene XML-Parser, unter anderem auch einen in der Java-Laufzeitbibliothek. Der Ausdruck

[29] XStream entwickelt sich weiter. Die Codebeispiele in diesem Text beruhen auf `xstream-1.4.1.jar`.

```
new XStream(new DomDriver())
```

erzeugt ein neues `XStream`-Objekt, das den Default-Parser der Laufzeitbibliothek verwendet. Das `DomDriver`-Objekt selbst ist lediglich ein Repräsentant des Parsers und spielt weiter keine Rolle.

Beispiel zur Serialisierung

Das folgende Programm `PlayersXstreamIO` serialisiert oder deserialisiert eine Liste mit zwei Spielern. Hier werden die gleichen Klassen `Player` (Listing 2.21) und `Players` (Listing 2.23) verwendet wie weiter vorne:

```
import com.thoughtworks.xstream.*;
import com.thoughtworks.xstream.io.xml.*;
import java.io.*;

public class PlayersXstreamIO {
    public static void main(String... args) throws IOException {
        XStream stream = new XStream(new DomDriver());

        Players players;
        if(args.length == 1)
            try(OutputStream output = new FileOutputStream(args[0])) {
                Player max = new Player("Max");
                max.setScore(5);
                Player moritz = new Player("Moritz");
                moritz.setScore(3);
                players = new Players().addPlayer(max).addPlayer(moritz);
                stream.toXML(players, output);
            }
        else
            try(InputStream input = new FileInputStream(args[0])) {
                players = (Players)stream.fromXML(input);
            }
        System.out.println(players);
    }
}
```

Listing 2.26: Schreiben und Lesen von Objekten mit XStream.

Das Programm wird mit einem oder zwei Kommandozeilenargumenten aufgerufen. Im ersten Fall serialisiert es eine Liste mit zwei Spielern auf die Datei, die als Argument angegeben ist. Im zweiten Fall restauriert es die Liste von dieser Datei.

```
$ java PlayersXstreamIO players.xml
[Max/5, Moritz/3]
$ java PlayersXstreamIO players.xml read
[Max/5, Moritz/3]
```

Gut lesbares Serialisierungsformat

Die Serialisierungsdatei `players.xml` enthält gut lesbares XML:[30]

```
$ cat players.xml
<Players>
  <players>
    <Player>
      <name>Max</name>
      <score>5</score>
    </Player>
    <Player>
      <name>Moritz</name>
      <score>3</score>
    </Player>
  </players>
</Players>
```

Klassennamen und Namen von Objektvariablen werden direkt in XML-Elementnamen umgesetzt. Dieses Dokument ist leichter lesbar als das Ergebnis der Serialisierung mit Beans (vergleiche Seite 159).

2.4.2 Referenzen

Umgang mit mehrfach serialisierten Objekten

XStream kommt mit mehrfachen Referenzen auf dasselbe Objekt, mit Referenzzyklen und mit Selbstreferenzen zurecht. Wird zum Beispiel ein `Players`-Objekt mit zweimal dem*selben* Spieler erzeugt

```
players = new Players().addPlayer(max).addPlayer(max);
```

und serialisiert, so erhält man die folgende XML-Ausgabe:

```
<Players>
  <players>
    <Player>
      <name>Max</name>
      <score>5</score>
    </Player>
    <Player reference="../Player"/>
  </players>
</Players>
```

Referenzen werden dabei durch sogenannte XPath-Ausdrücke umgesetzt, die ganz ähnlich wie Pfadangaben in einem Filesystem arbeiten. Die Referenz `../Player` ist beispielsweise zu lesen als: „Von diesem Punkt aus nach oben zum Elternelement, dann von dort aus zum (ersten) Element `Player`."

Alternative Darstellung von Referenzen

XPath-Ausdrücke sind in komplexen Strukturen nicht immer ganz einfach zu ver-

[30] In der Ausgabe fehlt die XML-Deklaration. Die Datei ist damit kein vollständiges XML-*Dokument*, sondern nur ein XML-*Element*. Sie ist als Baustein eines größeren Kontextes gedacht.

folgen. XStream kann wahlweise für Referenzen zwischen Elementen auch XML-Attribute namens `id` mit eindeutigen Nummern verwenden (siehe Seite 187). Dazu reicht es aus, das XStream-Objekt entsprechend zu konfigurieren:

```
xstream.setMode(XStream.ID_REFERENCES);
```

Die gleiche Spielerliste wie oben wird jetzt serialisiert als:

```
<Players id="1">
  <players id="2">
    <Player id="3">
      <name>Max</name>
      <score>5</score>
    </Player>
    <Player reference="3"/>
  </players>
</Players>
```

Selbstreferenzen, wie in der folgenden (sinnlosen) Klasse `SelfRef`, werden korrekt behandelt.

```
import java.beans.*;
import java.io.*;

public class SelfRef {
    private final SelfRef ref;

    public SelfRef() {
        ref = this;
    }
}
```

Listing 2.27: Serialisierung eines Objekts mit Selbstreferenz.

Ein `SelfRef`-Objekt wird beispielsweise serialisiert als:

```
<SelfRef>
  <ref reference=".."/>
</SelfRef>
```

Mit `id`-Attributen erhält man:

```
<SelfRef id="1">
  <ref reference="1"/>
</SelfRef>
```

2.4.3 Converter

Converter zur Serialisierung verschiedender Typen

XStream besteht im Kern aus einer Abbildungstabelle von Typen auf „Converter“. Converter sind Objekte, die für die Serialisierung beziehungsweise Deserialisierung der Werte bestimmter Typen zuständig sind. Alle Converter implementieren das Interface `Converter`.

Vordefinierte Converter

Für die primitiven Typen, Arrays und String stehen vordefinierte Converter zur Verfügung. Das Gleiche gilt für viele Klassen der Laufzeitbibliothek, wie zum Beispiel für Collections.

`ReflectionConverter` als Fallback

Objekte unbekannter Typen werden dem `ReflectionConverter` übergeben, der, wie der Name nahelegt, das betreffende Objekt mithilfe von Reflection (Kapitel 8) analysiert und die dabei gefundenen Bestandteile mit rekursiven Converter-Aufrufen verarbeitet.

2.4.4 Neue Converter, statische und transiente Variablen

Notwendigkeit eigener Converter

XStream ignoriert statische und transiente Variablen.[31] Letztere sind ohne Frage von der Serialisierung auszunehmen. Für Klassenvariablen lässt sich dagegen keine so pauschale Antwort geben. In vielen Fällen können sie nicht einfach übergangen werden, wie zum Beispiel beim `StampedCounter` (Listing 2.11).

Beispiel eines neuen Converters

Die Sammlung der vordefinierten Converter von XStream kann um neue Converter erweitert werden, die die Serialisierung bestimmter Typen anpassen oder komplett übernehmen. Als Beispiel soll die folgende Klasse `CountedPlayer` dienen. Sie ist von `Player` (Listing 2.21) abgeleitet und zählt die Anzahl der erzeugten Objekte in der Klassenvariablen `population` mit.

```
public class CountedPlayer extends Player {
    private static int population = 0;

    public CountedPlayer(String name) {
        super(name);
        population++;
    }
    public String toString() {
        return super.toString() + "(of " + population + ")";
    }
}
```

Listing 2.28: Spieler-Klasse, die die Anzahl der erzeugten Objekte mitzählt.

[31] XStream verhält sich damit genauso wie die binären Object-Streams.

Da Klassenvariablen bei der Serialisierung übergangen werden, steht der Objektzähler nach dem Deserialisieren wieder auf dem Startwert 0, obwohl es zwei Objekte gibt:

```
$ java CountedPlayersXstreamIO players.xml
[Max/5(of 2), Moritz/3(of 2)]
$ java CountedPlayersXstreamIO players.xml read
[Max/5(of 0), Moritz/3(of 0)]
```

Ableitung vom `ReflectionConverter`

Der neue Converter für `CountedPlayer` wird von `ReflectionConverter` abgeleitet und modifiziert dessen Verhalten. Als selbstständige Klasse hätte der neue Converter allerdings keinen Zugriff auf die private Klassenvariable `population`, um die er sich ja kümmern soll. Das Problem lässt sich mit einer statisch geschachtelten Klasse (Kapitel 5) in `CountedPlayer` beheben.[32] Diese Maßnahme hat keinen Einfluss auf die Arbeitsweise des neuen Converters, sondern vereinfacht nur das Zusammenspiel zwischen der Klasse und ihrem Converter. Die neue Klasse `Converter` wird also innerhalb von `CountedPlayer` definiert.

```
import com.thoughtworks.xstream.converters.*;
import com.thoughtworks.xstream.converters.reflection.*;
import com.thoughtworks.xstream.io.*;
import com.thoughtworks.xstream.mapper.*;

public class CountedPlayer extends Player {
    // ...

    public static class Converter extends ReflectionConverter {
        public Converter(Mapper mapper, ReflectionProvider provider) {
            super(mapper, provider);
        }

        public boolean canConvert(Class type) {
            // ...
        }

        protected void doMarshal(Object x,
                                 HierarchicalStreamWriter writer,
                                 MarshallingContext context) {
            // ...
            super.doMarshal(x, writer, context);
        }

        public Object doUnmarshal(Object x,
                                  HierarchicalStreamReader reader,
                                  UnmarshallingContext context) {
            // ...
            return super.doUnmarshal(x, reader, context);
```

[32] Der Name der geschachtelten Klasse kollidiert nicht mit dem gleichnamigen Interface `Converter`.

```
        }
    }

}
```

Listing 2.29: XStream-Custom-Converter als geschachtelte Klasse.

Der neue Converter braucht, abgesehen vom Konstruktor, drei Methoden:[33]

- `canConvert` gibt Auskunft, welche Typen dieser Converter verarbeitet. XStream übergibt dazu einen beliebigen Typ `type` und erwartet als Ergebnis `true` genau dann, wenn der Converter für diesen Typ zuständig ist.
- `doMarshal` wird zur Serialisierung aufgerufen. Ein Aufruf an die Basisklassenmethode löst das Defaultverhalten aus. Darüber hinaus wird die Klassenvariable `population` hier serialisiert.
- `doUnmarshal` wird bei der Deserialisierung aufgerufen. Auch hier wird an die Basisklassenmethode delegiert, aber noch zusätzlich `population` restauriert.

Wie der Wert der Klassenvariablen gerettet wird, ist alleine Sache von `doMarshal` und `doUnmarshal`. Im folgenden Beispiel wird ein neues XML-Element `population` eingeschoben und nachher wieder ausgelesen. `population` enthält den Wert der Klassenvariablen.

Eingriff in das XML-Dokument

Die beiden Methoden `startNode` und `endNode` öffnen und schließen beim Schreiben ein neues Element. `setValue` legt den Inhalt fest. Entsprechend öffnen und schließen `moveDown` und `moveUp` das nächste Element beim Einlesen. `getValue` liefert den Inhalt.

```
import com.thoughtworks.xstream.converters.*;
import com.thoughtworks.xstream.converters.reflection.*;
import com.thoughtworks.xstream.io.*;
import com.thoughtworks.xstream.mapper.*;

public class CountedPlayer extends Player {
    // ...

    public static class Converter extends ReflectionConverter {
        public Converter(Mapper mapper, ReflectionProvider provider) {
            super(mapper, provider);
        }

        public boolean canConvert(Class type) {
            return type == CountedPlayer.class;
        }
```

[33] Die Methoden `doMarshal` und `doUnmarshal` gehen genauso vor wie `writeObject` und `readObject` bei den Object-Streams (Seite 144).

```
        protected void doMarshal(Object x,
                                 HierarchicalStreamWriter writer,
                                 MarshallingContext context) {
            writer.startNode("population");
            writer.setValue(Integer.toString(population));
            writer.endNode();
            super.doMarshal(x, writer, context);
        }

        public Object doUnmarshal(Object x,
                                  HierarchicalStreamReader reader,
                                  UnmarshallingContext context) {
            reader.moveDown();
            population = Integer.parseInt(reader.getValue());
            reader.moveUp();
            return super.doUnmarshal(x, reader, context);
        }
    }

}
```

Listing 2.30: Methoden des XStream-Custom-Converter.

Der neue Converter wird im Hauptprogramm mit einem Aufruf der Methode `registerConverter` registriert.[34]

Registrieren eines neuen Converters

```
import com.thoughtworks.xstream.*;
import com.thoughtworks.xstream.io.xml.*;
import java.io.*;

public class CountedPlayersXstreamIO {
    public static void main(String... args) throws IOException {
        XStream stream = new XStream(new DomDriver());
        stream.registerConverter(new CountedPlayer.Converter(stream.getMapper(),
                                 stream.getReflectionProvider()));
        Players players;
        if(args.length == 1)
            try(OutputStream output = new FileOutputStream(args[0])) {
                Player max = new CountedPlayer("Max");
                max.setScore(5);
                Player moritz = new CountedPlayer("Moritz");
                moritz.setScore(3);
                players = new Players().addPlayer(max).addPlayer(moritz);
                stream.toXML(players, output);
            }
        else
            try(InputStream input = new FileInputStream(args[0])) {
                players = (Players)stream.fromXML(input);
            }
        System.out.println(players);
    }
```

[34] Die etwas eigenartige Syntax `new CountedPlayer.Converter(...)` des Konstruktoraufrufs ist wegen der Definition als statisch geschachtelte Klasse nötig.

```
}
```

Listing 2.31: Ein- und Ausgabe mit einem XStream-Custom-Converter.

Der neue Converter restauriert die Objekte samt der Klassenvariablen korrekt:

```
$ java CountedPlayersXstreamIO players.xml
[Max/5(of 2), Moritz/3(of 2)]
$ java CountedPlayersXstreamIO players.xml read
[Max/5(of 2), Moritz/3(of 2)]
```

In der serialisierten Datei sind jetzt die zusätzlichen `population`-Elemente zu sehen:

```
<Players>
  <players>
    <CountedPlayer>
      <population>2</population>
      <name>Max</name>
      <score>5</score>
    </CountedPlayer>
    <CountedPlayer>
      <population>2</population>
      <name>Moritz</name>
      <score>3</score>
    </CountedPlayer>
  </players>
</Players>
```

Verbesserungen des Beispiels

Diese Lösung hat noch Schönheitsfehler. Zum einen wird der Zähler beim Deserialisieren überschrieben, auch wenn vor der Deserialisierung schon Objekte mit normalen Konstruktoraufrufen erzeugt wurden. Zum anderen wird das gleiche Element in *jeden* serialisierten `CountedPlayer` eingefügt und ist damit redundant. Eine einfachere und robustere Version würde einfach in `doUnmarshal` die Klassenvariable inkrementieren und damit den Effekt von regulären Konstruktoraufrufen nachvollziehen.

2.4.5 Encoding

Steuerung des Encodings

XStream liefert keine kompletten XML-*Dokumente*, sondern nur isolierte *Elemente*. Insbesondere fehlt die XML-Deklaration, deren an dieser Stelle wichtigster Beitrag die Angabe des Encodings ist (siehe Seite 179).

Das Encoding der serialisierten Datei wird nicht von XStream selbst abgewickelt, sondern dem zugrunde liegenden I/O-Objekt überlassen. Ruft man `toXML` und `fromXML` mit einem passend konstruierten `Writer` oder `Reader` auf, so wird das gewünschte Encoding verwendet. Weitere Einzelheiten finden Sie auf Seite 87. Das folgende Programm erzeugt oder liest eine XML-Datei in dem Encoding, das als zweites Kommandozeilenargument angegeben ist.

```
import com.thoughtworks.xstream.*;
import com.thoughtworks.xstream.io.xml.*;
import java.io.*;

public class PlayersXstreamEncodingIO {
    public static void main(String... args) throws IOException {
        XStream stream = new XStream(new DomDriver());

        Players players;
        if(args.length == 2)
            try(OutputStream output = new FileOutputStream(args[0]);
                Writer writer = new OutputStreamWriter(output, args[1])) {
                Player max = new Player("Mäx");
                max.setScore(5);
                Player moritz = new Player("Moritz");
                moritz.setScore(3);
                players = new Players().addPlayer(max).addPlayer(moritz);
                stream.toXML(players, writer);
            }
        else
            try(InputStream input = new FileInputStream(args[0]);
                Reader reader = new InputStreamReader(input, args[1])) {
                players = (Players)stream.fromXML(reader);
            }
        System.out.println(players);
    }
}
```

Listing 2.32: Ein- und Ausgabe mit XStream mit wählbarem Encoding.

Der Name des ersten Spielers (Mäx) enthält einen Umlaut, der korrekt serialisiert und restauriert wird.

2.4.6 Kompatibilität mit Object-Streams

XStream stellt Objekte zur Verfügung, die zu `ObjectOutputStream` und `ObjectInputStream` (Seite 127) kompatibel sind:

Ähnlichkeit mit Object-Streams

```
ObjectOutputStream objectOutput = xstream.createObjectOutputStream(writer);
ObjectInputStream objectInput = xstream.createObjectInputStream(reader);
```

Zur weiteren Verarbeitung stehen die gleichen Mechanismen zur Verfügung, wie mit den Klassen aus `java.io`. Unabhängig davon wird aber weiterhin XML als Serialisierungsformat verwendet. Das folgende Programm `ObjectXstreamIO` ist praktisch identisch zu `ObjectIO` (Listing 2.1), lediglich die Initialisierung der Object-Streams wird über XStream abgewickelt.

```
import com.thoughtworks.xstream.*;
import com.thoughtworks.xstream.io.xml.*;
import java.io.*;
import java.nio.file.*;
import java.util.*;

public class ObjectXstreamIO {
    public static void main(String... args) throws IOException, ClassNotFoundException {
        XStream stream = new XStream(new DomDriver());
        if(args.length == 1)
            // output
            try(OutputStream output = Files.newOutputStream(Paths.get(args[0]));
                ObjectOutputStream objectOutput = stream.createObjectOutputStream(output)
                objectOutput.writeObject(new Date());
                objectOutput.writeObject(new Random());
            }
        else
            // input
            try(InputStream input = Files.newInputStream(Paths.get(args[0]));
                ObjectInputStream objectInput = stream.createObjectInputStream(input)) {
                System.out.println(objectInput.readObject());
                Random random = (Random)objectInput.readObject();
                System.out.println(random.nextInt());
            }
    }
}
```

Listing 2.33: Lesen und Schreiben von Objekten mit XStream.

Der Aufruf ist vollkommen entsprechend zu `ObjectIO`. Die serialisierte Datei enthält jetzt ein XML-Element statt binärer Daten.

```
<object-stream>
  <date>2011-04-16 13:33:56.526 UTC</date>
  <java.util.Random serialization="custom">
    <java.util.Random>
      <default>
        <seed>85258001382955</seed>
        <nextNextGaussian>0.0</nextNextGaussian>
        <haveNextNextGaussian>false</haveNextNextGaussian>
      </default>
    </java.util.Random>
  </java.util.Random>
</object-stream>
```

Auf diese Art lassen sich Programme, die ursprünglich für Object-Streams entwickelt wurden, mit wenig Aufwand auf XML als Serialisierungsformat umstellen.

Zusammenfassung

- Die Methoden `writeObject` und `readObject` der Klassen `ObjectOutputStream` und `ObjectInputStream` können Java-Objekte in und aus Byteströmen **serialisieren** beziehungsweise **deserialisieren**.
- Das **Interface** `Serializable` entscheidet über die Serialisierbarkeit einer Klasse.
- Die Deserialisierung kann Objekte **ohne Konstruktoraufrufe** erzeugen.
- Der **Modifier** `transient` nimmt Objektvariablen von der Serialisierung aus.
- Die Klassenvariable `serialVersionUID` steuert das Verhalten bei **veränderten Klassendefinitionen**.
- `XMLEncoder` und `XMLDecoder` serialisieren beziehungsweise deserialisieren **JavaBeans** in **XML-Dateien**.
- `PersistenceDelegate`**s** beziehen Klassen ein, die nicht den Beans-Konventionen folgen.
- So generierte XML-Dateien repräsentieren **Java-Code zur Rekonstruktion** von Objekten.
- Die externe Bibliothek **XStream** kombiniert Vorteile der binären Serialisierung mit denen von JavaBeans.
- **XStream-Converter** regeln im Detail die Abbildung zwischen Objekten und XML-Strukturen.

Aufgaben

Aufgabe 1: Serialisierte Pizzas

In Abschnitt C werden zur Illustration des Decorator-Patterns Pizzas modelliert. Eine Pizza besteht technisch aus mehreren verknüpften Objekten.

Ändern Sie im Quelltext *eine einzige Zeile*, um Pizzas serialisierbar zu machen.

Das folgende Testprogramm funktioniert ähnlich wie `PizzaMain`. Allerdings serialisiert es eine neu aufgebaute Pizza auf die Standardausgabe. Wenn es ohne Kommandozeilenargumente aufgerufen wird, liest es eine serialisierte Pizza von der Standardeingabe:

```
import java.io.*;

public class PizzaIO {
    public static void main(String... args) throws IOException, ClassNotFoundException {
        Pizza pizza = null;
        if(args.length == 0)
            try(ObjectInputStream objectInput = new ObjectInputStream(System.in)) {
                pizza = (Pizza)objectInput.readObject();
            }
        else
            try(ObjectOutputStream objectOutput = new ObjectOutputStream(System.out)) {
                for(String arg: args)
                    // ... wie PizzaMain
                objectOutput.writeObject(pizza);
            }
        System.err.printf("Your pizza:%nprice: %d%nvegetarian: %b%nhot: %b%n",
                          pizza.getPrice(),
                          pizza.isVegetarian(),
                          pizza.isHot());
    }
}
```

Listing 2.34: Serialisiert oder deserialisiert eine Pizza via Object-Streams.

Sorgen Sie dafür, dass sich Pizzas serialisieren lassen.

Testen Sie dann die Serialisierung folgendermaßen:

```
$ java PizzaIO Crunchy Cheese Cheese Salami Chili > pizza.ser
Your pizza:
price: 650
vegetarian: false
hot: true
$ java PizzaIO < pizza.ser
Your pizza:
price: 650
vegetarian: false
hot: true
```

Gehen Sie den folgenden Fragen nach und erklären Sie die Beobachtung.

- Was wird aus serialisierten Objekten, wenn der Rumpf von Methoden nachträglich modifiziert wird? Ändern Sie den Preis in `Salami` von 1,50 auf 2,00 Euro und lesen Sie dann eine vor der Änderung serialisierte Pizza ein.
- Lässt sich eine Datei mit serialisierten Objekten noch lesen, wenn am Ende beliebige Daten angefügt werden? Verlängern Sie `pizza.ser` mit Mitteln des Betriebssystems um beliebige Daten.
- Welche UID hat der Aufzählungstyp `Base`?

Definieren Sie das besondere Aroma „Sunday-Delight“ als neue Pizzaauflage. Sunday-Delight darf nur an Sonntagen verwendet werden, kostet nichts, ist rein pflanzlich und nicht scharf.

Der folgende Test stellt sicher, dass der Code am Sonntag läuft:

```
if(new GregorianCalendar().get(Calendar.DAY_OF_WEEK) != Calendar.SUNDAY)
    throw new AssertionError("running on Sunday only");
```

Sorgen Sie dafür, dass `SundayDelight`-Objekte nur an Sonntagen entstehen. Greifen Sie im Konstruktor und in die Deserialisierung ein.[35] Bei dem Versuch, an einem anderen Tag als sonntags eine Pizza mit Sunday-Delight zu erzeugen, bricht das Programm ab:

```
$ java PizzaIO Crunchy Cheese Cheese Salami SundayDelight Chili > pizza.ser
Exception in thread "main" java.lang.AssertionError: no Delight except on Sunday
```

Aufgabe 2: Pizzas als JavaBeans

Modifizieren Sie die Typen des Abschnitts C so, dass Pizzas als JavaBeans in XML-Dokumente serialisiert und daraus rekonstruiert werden können. Dazu ist eine einzige zusätzliche Methode erforderlich.

Das folgende Programm setzt die modifizierten Klassen ein. Es funktioniert wie `PizzaIO` (Listing 2.34), benutzt aber den XML-Encoder und -Decoder aus `java.beans`:

```
import java.beans.*;
import java.io.*;

public class PizzaBeanIO {
    public static void main(String... args) throws IOException, ClassNotFoundException {
        Pizza pizza = null;
        if(args.length == 0)
            try(XMLDecoder decoder = new XMLDecoder(System.in)) {
                pizza = (Pizza)decoder.readObject();
            }
        else
            try(XMLEncoder encoder = new XMLEncoder(System.out)) {
                for(String arg: args)
                    // ... wie PizzaMain
```

[35] Das Problem, dass ein Programm mit `SundayDelight`-Objekten am Sonntag gestartet wird und am Montag noch läuft, sei hier zurückgestellt.

```
                encoder.writeObject(pizza);
            }
        System.err.printf("Your pizza:%nprice: %d%nvegetarian: %b%nhot: %b%n",
                          pizza.getPrice(),
                          pizza.isVegetarian(),
                          pizza.isHot());
    }
}
```

Listing 2.35: Serialisiert oder deserialisiert eine Pizza als JavaBean.

Dieses Programm ist noch unvollständig und muss ergänzt werden, weil die verschiedenen Klassen nicht den Beans-Konventionen folgen und daher eigene Serialisierer brauchen (siehe Abschnitt 2.3.6).

`PizzaBeanIO` schreibt ein XML-Dokument mit einer serialisierten Pizza auf die Standardausgabe, die hier in einer Datei aufgefangen wird:

```
$ java PizzaBeanIO Crunchy Cheese Cheese Salami Chili > pizza.xml
Your pizza:
price: 650
vegetarian: false
hot: true
```

Beim Aufruf ohne Argumente restauriert `PizzaBeanIO` die Pizza wieder von der Standardeingabe:

```
$ java PizzaBeanIO < pizza.xml
Your pizza:
price: 650
vegetarian: false
hot: true
```

Die XML-Datei enthält die Objektstruktur in gut lesbarer Form:

```
$ cat pizza.xml
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.7.0_03" class="java.beans.XMLDecoder">
    <object class="Chili">
        <object class="Salami">
            <object class="Cheese">
                <object class="Cheese">
                    <object class="Crunchy"/>
                </object>
            </object>
        </object>
    </object>
</java>
```

Das XML-Dokument beschreibt den folgenden Code, den die Deserialisierung ausführt:

```
new Chili(
    new Salami(
        new Cheese(
            new Cheese(
                new Crunchy()))))
```

Es kann mit einem Texteditor modifiziert werden. Fügen Sie eine zusätzliche Auflage Salami ein und ändern Sie den Pizzaboden auf „Puffy“. Lesen Sie das modifizierte XML-Dokument wieder ein:

```
$ java PizzaBeanIO < pizza.xml
Your pizza:
price: 900
vegetarian: false
hot: true
```

Aufgabe 3: Pizzas via XStream

Wie in Abschnitt 2.4.6 erklärt, kann die XStream-Bibliothek weitgehend kompatibel zur regulären Objektserialisierung arbeiten. Passen Sie das Programm `PizzaIO` (Listing 2.34) entspreched an. Die Änderungen beschränken sich auf drei Zeilen.

Das modifizierte Programm funktioniert wie vorher:

```
$ java PizzaXstreamIO Crunchy Cheese Cheese Salami Chili > pizza.xml
Your pizza:
price: 650
vegetarian: false
hot: true
$ java PizzaXstreamIO < pizza.xml
Your pizza:
price: 650
vegetarian: false
hot: true
```

Es schreibt und liest aber XML-Dokumente statt binärer Dateien:

```
$ cat pizza.xml
<object-stream>
```

```
    <Chili>
        <below class="Salami">
            <below class="Cheese">
                <below class="Cheese">
                    <below class="Crunchy">
                        <price>300</price>
                        <hot>false</hot>
                    </below>
                </below>
            </below>
        </below>
    </Chili>
</object-stream>
```

Überprüfen Sie, wie XStream mit der explizit geregelten Deserialisierung von `SundayDelight` umgeht, die nur an Sonntagen funktioniert.

Kapitel 3 XML

Lernziele

In diesem Kapitel lernen Sie

- wie system- und programmiersprachenunabhängige **XML-Dokumente** aufgebaut sind und wie man ihre Struktur festlegen und mit Validatoren überprüfen kann (Abschnitt 3.1).
- wie ein Java-Programm ein XML-Dokument mit einem **DOM-Parser** laden, auswerten, modifizieren, schreiben und auch neu erzeugen kann (Abschnitt 3.2).
- wie ein Java-Programm mit einem **SAX-Parser** ein XML-Dokument schon während des Lesens verarbeiten und dadurch schneller und sparsamer als ein DOM-Parser arbeiten kann (Abschnitt 3.3).

Die „Extended Markup Language“ XML ist ein system- und programmiersprachenneutrales Textformat zur Darstellung fast beliebiger Informationen. Nützlich ist XML vor allem zum Datenaustausch zwischen Programmen und Rechnern und zum langfristigen Speichern von Daten. Als Textformat kann es zwar von Menschen gelesen und geschrieben werden, ist aber trotzdem in erster Linie für Maschinen gedacht.

3.1 Struktur, Grammatik und Validierung

3.1.1 Aufbau eines XML-Dokuments

Öffentliche Spezifikationen

Einige Spezifikationen des „World Wide Web Consortium“ W3C regeln den genauen Aufbau von XML und der damit zusammenhängenden Technologien. Verschiedene Formate dieser Spezifikationen sind auf der Webseite des W3C (`http://www.w3.org/standards/xml`) öffentlich verfügbar. Für dieses Kapitel maßgeblich ist der Standard „Extensible

Markup Language (XML) 1.0 (Fifth Edition)“ [1][1] vom 26.11.2008. Eng damit zusammen hängt der weitere Standard „Namespaces in XML 1.0 (Third Edition)“ [2], der Namespaces (siehe Seite 182) definiert.

Knoten und Elemente

Baumstruktur, Elemente und Textknoten

Aus logischer Sicht ist ein XML-Dokument eine Textdarstellung eines Baums.[2] Der Baum besteht aus verschiedenen Arten von Knoten. Die wichtigsten Arten sind Elemente und Textknoten:

- **Elemente** sind „innere Knoten“ des Baums. Sie haben eine beliebige Anzahl von Kindknoten in einer festen Reihenfolge. Elemente, die keine Kinder haben, sind Blätter des Baums.
- **Textknoten** bestehen nur aus einer Zeichenkette. Textknoten sind immer Blätter.
- **Kommentare** in XML-Dokumenten werden bei der Verarbeitung aber in der Regel ignoriert. Die Schreibweise von XML-Kommentaren ist:[3]

  ```
  <!-- ... -->
  ```

  Kommentare dürfen vor und nach jedem Knoten stehen.

Elementnamen und Tags

Elemente haben **Namen**, die ähnlichen Regeln wie Java-Bezeichner folgen.[4] In der Textdarstellung besteht ein Element aus einem **öffnenden Tag**[5] und einem **schließenden Tag**, zwischen denen die Kindknoten aufgezählt sind. Beide Tags sind jeweils mit spitzen Klammern eingefasst, um sie eindeutig erkennen zu können. Das zweite Zeichen eines schließenden Tags ist `/`, das zweite Zeichen eines öffnenden Tags nicht.

```
<element> ... Kindknoten ... </element>
```

Elemente ohne Kindknoten können kürzer als **leeres Tag** geschrieben werden. Die beiden folgenden Schreibweisen sind gleichwertig:

---

1 Der Text weist sich selbst als „Recommendation“ (Empfehlung) aus. Der Begriff klingt etwas zurückhaltend. Tatsächlich ist der Status einer „Recommendation“ ziemlich verbindlich und weitgehend stabil.

2 Der Baum ist n-stellig und geordnet. Das heißt, dass jeder Knoten beliebig viele Kindknoten haben kann und dass die Reihenfolge der Kindknoten signifikant ist.

3 Abgesehen von den Enden dürfen innerhalb eines Kommentars keine zwei Bindestriche direkt aufeinanderfolgen.

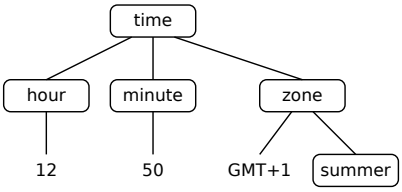
4 Das erste Zeichen eines Elementnamens darf ein Unicode-Buchstabe, ein Doppelpunkt und ein Unterstrich sein. Als Folgezeichen sind zusätzlich Bindestrich, Punkt und Dezimalziffern erlaubt. Groß- und Kleinschreibung sind signifikant.

5 Der englische Begriff *tag* bedeutet „Etikett“ oder „Anhängeschildchen“.

```
<element/>
<element></element>
```

Textknoten werden wörtlich als Zeichenkette geschrieben.[6] Textknoten

Die folgende Skizze zeigt die Baumstruktur einer Zeitangabe. Elemente sind als abgerundete Kästchen dargestellt, Textknoten nur als Zeichenkette:



Dieser Baum entspricht der folgenden Textdarstellung:[7]

```
<time>
   <hour>12</hour>
   <minute>50</minute>
   <zone>
       GMT+1
       <summer/>
   </zone>
</time>
```

Listing 3.1: Geschachtelte XML-Elemente mit einer Zeitangabe.

Dokument und Deklaration

Ein komplettes XML-Dokument besteht aus einer XML-Deklaration und genau einem Element, dem „Wurzelelement“ (*root element*). Die **XML-Deklaration** hat die Syntax XML-Deklaration als Kopf eines XML-Dokuments

```
<?xml version="version" encoding="encoding"?>
```

- Die Versionsangabe ist verpflichtend. Derzeit gibt es die XML-Versionen 1.0 und 1.1. Der Unterschied zwischen den beiden Versionen spielt hier aber keine Rolle.

6 Einzelne Textzeichen müssen unter Umständen maskiert werden, wie weiter unten erklärt wird.

7 Genau genommen enthält die Textdarstellung zusätzliche Zeilenumbrüche und Einrückungen, die im Baum fehlen. Ob derartiger Zwischenraum signifikant ist oder nicht, hängt von der konkreten Anwendung ab.

- Die Angabe des Encodings ist optional. Sie ist nötig, wenn andere als nur ASCII-Zeichen im Dokument vorkommen. Häufig benutzte Encodings sind `utf-8` und `iso-8859-1`.[8]

Das Element in der Textdatei `time-element.txt` (Listing 3.1) ergibt zusammen mit einer XML-Deklaration als Kopfzeile ein vollständiges XML-Dokument:

```
<?xml version="1.0"?>
<time>
   <hour>12</hour>
   <minute>50</minute>
   <zone>
       GMT+1
       <summer/>
   </zone>
</time>
```

Listing 3.2: `time.xml`, XML-Dokument mit einer Zeitangabe.

Attribute

Attribute mit Zusatzinformationen für Knoten

Elemente können mit **Attributen** näher beschrieben werden. Ein Attribut besteht aus einem Namen und dem Wert. Attributnamen folgen ähnlichen Regeln wie Elementnamen, Attributwerte können beliebige Strings sein.[9] Attribute werden im öffnenden beziehungsweise leeren Tag aufgezählt:

```
<element attribute1="value1" attribute2="value2" ...>...</element>
<element attribute1="value1" attribute2="value2" .../>
```

Die Attributnamen eines Elements müssen eindeutig sein. Die Reihenfolge ist dagegen ohne Bedeutung.

Attribute sollen nähere Angaben *über* den Inhalt eines Elements beisteuern (Meta-Informationen). Der Inhalt selbst findet sich in den Kindknoten und nicht in Attributen.

Attribute vs. Inhalt von Knoten

Manchmal fällt die Unterscheidung von Inhalt und Meta-Informationen nicht ganz leicht. Bei strukturierten, zusammengesetzten Daten stellt sich die Frage nicht, weil Attributwerte nur flache Zeichenketten aufnehmen können.[10]

8 Bei fehlender Angabe gilt automatisch `utf-8`.

9 Bestimmte Zeichen müssen gegebenenfalls umschrieben werden, wie weiter unten erklärt wird.

10 Man könnte eine eigene Syntax für Attributwerte definieren und dann mit einem eigenen Parser strukturierte Daten aus dem Text gewinnen. Dass das am Zweck von Attributen vorbeiführt, liegt auf der Hand.

Bei einfachem Text hat man allerdings die Wahl. Er ließe sich sowohl als Inhalt in einem Text-Kindknoten verstauen, als auch in einem Attributwert. Hilfreich ist manchmal die Überlegung, ob der fragliche Text essenziell für das Element ist oder ob das Element auch ohne die Information auskommen könnte. Eine Zeitangabe ohne Zeitzone kann durchaus sinnvoll sein, eine Zeitangabe ohne Stundenzahl dagegen nicht. Deshalb sollte „GMT+1“ in einem Attributwert stehen und „12“ in einem Textknoten. Die Grenze ist allerdings unscharf und hängt zudem vom Anwendungszweck ab. Im Zweifelsfall sind beide Varianten akzeptabel, wenn sich nur alle Beteiligten einig sind.

Im folgenden Beispiel ist die Angabe der Zeitzone in Attribute verschoben.

```
<?xml version="1.0"?>
<time zone="GMT+1" summer="true">
   <hour>12</hour>
   <minute>50</minute>
</time>
```

Listing 3.3: `time-attribute.xml`, Angaben zum Teil in XML-Attributen.

Attribute ohne Wert

Das Element `summer` war in der ursprünglichen Fassung `time.xml` (Listing 3.2) leer, es hatte keinen Inhalt. Maßgeblich war bloß die Existenz oder Abwesenheit. Überträgt man diese Interpretation auf das *Attribut* `summer`, dann sollte auch hier nur die Existenz zählen. Die Syntax von Attributen schreibt allerdings einen Wert vor. Der String `true` erfüllt diese Forderung.[11]

In diesem einfachen Beispiel könnte sogar die gesamte Information in Attribute gepackt werden. Übrig bleibt ein leeres Element mit vier Attributen:

```
<?xml version="1.0"?>
<time hour="12" minute="50" zone="GMT+1" summer="true"/>
```

Listing 3.4: `time-attributes-only.xml`, alle Angaben in XML-Attributen.

Dieses XML-Dokument ist syntaktisch korrekt. Ob die Struktur allerdings sinnvoll ist, sei dahingestellt.

Ersatzdarstellungen

Meta-Symbole in Attributwerten

Tags beginnen mit einer öffnenden spitzen Klammer. Allerdings kann < auch als

[11] Das führt zu der Frage, wie bei alleiniger Berücksichtigung der Anwesenheit eigentlich das Attribut `summer="false"` zu lesen wäre. Weiter unten wird diese Angabe als unzulässig erklärt, ebenso wie etwa `summer="3.141592"`.

Textzeichen vorkommen. Für diesen Fall sind die folgenden Ersatzdarstellungen vorgesehen:

<	<
&	&
>	>

Text mit vielen Meta-Symbolen

Diese Ersatzdarstellungen reichen für einzelne Vorkommen der Zeichen aus. Bei Häufungen führen sie aber zu aufgeblähtem und schwer lesbarem Text. Mit einem „CDATA-Abschnitt“ können ganze Textpassagen komplett „entwertet“ werden:

```
<![CDATA[text]]>
```

Der gesamte Text zwischen den inneren eckigen Klammern gilt wörtlich, einschließlich aller Vorkommen von <, & und >. Die beiden folgenden Zeilen sind gleichwertig:[12]

```
<![CDATA[&amp; ist die Ersatzdarstellung von &.]]>
&amp;amp; ist die Ersatzdarstellung von &amp.
```

Begrenzer im Inhalt

Ein ähnliches Problem werfen Attributwerte auf. Als Begrenzer stehen Gänsefüßchen und Apostrophe zur Wahl.[13] Die folgenden Ersatzdarstellungen erlauben beide Zeichen in den Inhalt aufzunehmen:

’	'
"	"

Die folgenden Zeilen sind äquivalent:

```
<rezept quelle="Sammlung: &quot;Oma&apos;s Geheimnisse&quot"/>...
<rezept quelle='Sammlung: &quot;Oma&apos;s Geheimnisse&quot'/>...
```

Namespaces

Kollisionen von Element- und Attributnamen

XML-Dokumente können Teile enthalten, die aus anderen Quellen stammen.[14] Dabei können zufällig gleiche Element- und Attributnamen aufeinandertreffen, die nichts miteinander zu tun haben.

[12] Der Vollständigkeit wegen sei erwähnt, dass auch in einem CDATA-Abschnitt für das Zeichen > die Ersatzdarstellung nötig ist, wenn es als *Inhalt* direkt nach]] folgt.

[13] Allerdings muss das gleiche Begrenzerzeichen am Anfang und am Ende stehen.

[14] Ein praktisches Beispiel sind XHTML-Seiten mit Abbildungen im SVG-Format und Formeln im MathML-Format.

Ähnliche Konflikte ergeben sich in Java mit Klassen gleichen Namens aus unabhängigen Bibliotheken. In Java beseitigen Packages die Kollisionen. In XML lösen **Namespaces** das Problem. Trotz der grundsätzlich gleichen Idee haben XML-Namespaces aber andere Bindungsregeln und Gültigkeitsbereiche als Java-Packages.

Namespaces zur Beseitigung von Kollisionen

Attribut- und Elementnamen in XML-Dokumenten können mit einem Namespace-Präfix versehen werden, das mit einem Doppelpunkt abgetrennt vor den undekorierten Namen gesetzt wird:

```
<namespace:element>... Kindknoten ...</namespace:element>
```

Gleiche Elementnamen mit unterschiedlichen Präfixen kommen sich nicht ins Gehege. Im Beispiel

```
<time>
   <clock:hour>12</clock:hour>
   <clock:minute>50</clock:minute>
   <zone>
       <timezone:name>GMT+1</timezone:name>
       <timezone:summer/>
   </zone>
</time>
```

Listing 3.5: Elemente mit Namespace-Präfixen.

gelten die folgenden Namespace-Zuordnungen:

```
time       -
hour       clock
minute     clock
zone       -
name       timezone
summer     timezone
```

Zugriff auf Namespaces

Eine **Namespace-Deklaration** macht einen Namespace verfügbar, ähnlich wie eine Import-Klausel ein Java-Package öffnet. Syntaktisch hat eine Namespace-Deklaration die Gestalt eines Attributs, dessen Name selbst mit dem Präfix `xmlns` versehen ist:[15]

```
xmlns:prefix="url"
```

Die Deklaration verbindet eine URL mit einem Präfix.

[15] Es gibt keine Deklaration des Namespace `xmlns`. Er steht automatisch zur Verfügung.

- Die URL identifiziert den Namespace. Tatsächlich wird die URL nicht unbedingt benutzt, sondern dient nur zur eindeutigen Kennzeichnung. In der Praxis ist es allerdings sinnvoll und üblich, auf der entsprechenden Webseite weiterführende Informationen zum Namespace zu hinterlegen.
- Das Präfix ist ein Kürzel für den Namespace in diesem Dokument. Es kann beliebig gewählt[16] werden und muss insbesondere nichts mit der URL zu tun haben. In einem anderen Dokument kann derselbe Namespace mit einem anderen Präfix versehen werden.

Ein Namespace kann im attributierten Element selbst und in allen direkten und indirekten Kindelementen verwendet werden. Jedes Vorkommen betroffener Namen muss allerdings einzeln mit dem Präfix versehen werden, zusätzlich zur Deklaration. Ein Name kann nur einem Namespace zugeordnet sein, nicht mehreren.

Im Beispiel

```
<?xml version="1.0"?>
<time xmlns:clock="http://sol.cs.hm.edu/xmlnamespaces/clock">
   <clock:hour>12</clock:hour>
   <clock:minute>50</clock:minute>
   <zone xmlns:timezone="http://sol.cs.hm.edu/xmlnamespaces/timezone">
       <timezone:name>GMT+1</timezone:name>
       <timezone:summer/>
   </zone>
</time>
```

Listing 3.6: Namespaces mit Namespace-Deklarationen.

sind die folgenden Präfixe festgelegt:

```
timezone   http://sol.cs.hm.edu/xmlnamespaces/timezone
clock      http://sol.cs.hm.edu/xmlnamespaces/clock
```

Voreingestellter Namespace für Namen ohne Angabe

Ein **Default-Namespace** verbessert die Lesbarkeit. Er gilt für alle untergeordneten Elemente und Attribute ohne Präfix. Syntaktisch fehlt in der Deklaration des Default-Namespace das Präfix:

```
xmlns="url"
```

Im folgenden Beispiel liegen `zone`, `name` und `summer` im Default-Namespace `http://sol.cs.hm.edu/xmlnames`

[16] Verboten sind nur die Anfangsbuchstaben „xml“.

```
<?xml version="1.0"?>
<time>
   <hour>12</hour>
   <minute>50</minute>
   <zone xmlns="http://sol.cs.hm.edu/xmlnamespaces/timezone">
       <name>GMT+1</name>
       <summer/>
   </zone>
</time>
```

Listing 3.7: Deklaration eines Default-Namespace.

Elementnamen in keinem Namespace

Elemente eines Dokuments außerhalb einer Namespace-Deklaration liegen in keinem Namespace. Das gilt beispielsweise im vorhergehenden Beispiel für `time`, `hour` und `minute`, aber auch für Dokumente ganz ohne Namespace-Deklaration.[17]

Wohlgeformtes XML-Dokument

Etwas anders verhalten sich Attribute ohne Präfix: Hier wird statt des Default-Namespace der Namespace des attributierten Elements übernommen. Ein XML-Dokument, das den hier beschriebenen syntaktischen Regeln folgt, ist **wohlgeformt** (*well-formed*).

3.1.2 Validierung

Definition der Struktur von XML-Dokumenten

Wohlgeformtheit ist verpflichtend für XML-Dokumente.[18] Darüber hinaus müssen sich alle Beteiligten über die verwendeten Bausteine und deren Anordnung in einem XML-Dokument einig sein, um zu einer einheitlichen Interpretation des Inhalts zu kommen.

Diese strukturellen Regeln werden als XML-Vokabular oder **XML-Grammatik** bezeichnet.[19] Eine XML-Grammatik regelt die

- zulässigen Element- und Attributnamen,
- Schachtelung von Elementen und Textknoten,
- Platzierung von Attributen,
- möglichen Inhalte von Attributwerten und Textknoten.

Verbreitete XML-Grammatiken

Es gibt nicht nur eine XML-Grammatik, sondern beliebig viele. Jede Anwendung

[17] Eine Namespace-Deklaration ohne Präfix und URL hat die gleiche Wirkung. Untergeordnete Elemente ohne Präfix liegen in keinem Namespace. Ein gegebenenfalls gültiger Default-Namespace wird ausgeblendet.

[18] Es gibt kein „nicht wohlgeformtes“ XML-Dokument. Ein solcher Text wäre überhaupt kein XML-Dokument.

[19] Streng genommen gibt es auf einer tieferen Ebene eine weitere Grammatik von XML, die im vorhergehenden Abschnitt beschrieben wurde. Diese zugrunde liegende Grammatik stellt Wohlgeformtheit sicher.

kann eine eigene XML-Grammatik erfinden, passend zu den konkreten Anforderungen. Es gibt bereits viele populäre XML-Grammatiken, wie zum Beispiel

- XHTML[20] für Webseiten,
- SVG für Zeichnungen,
- MathML zur Repräsentation mathematischer Formeln,
- RSS als Nachrichtenformat für Feed-Reader und
- XML-Schema[21] zur Definition von XML-Grammatiken.

Valide XML-Dokumente

Ein wohlgeformtes XML-Dokument, das einer bestimmten XML-Grammatik folgt, ist **valide** bezüglich dieser Grammatik. „Validierung“ ist der Test auf Konformität gegenüber einer gegebenen Grammatik. Validität ohne Wohlgeformtheit ist gegenstandslos.

Es gibt eine ganze Reihe von Ansätzen zur Definition von XML-Grammatiken. Zwei Vertreter mit sehr unterschiedlichen Eigenschaften, DTD und XML-Schema, werden hier vorgestellt.

DTD

Schreibweise für XML-Grammatiken

„Document Type Definitions“ (DTDs) sind einer der ersten Formalismen für XML-Grammatiken. DTDs definieren zwar XML-Grammatiken, sind selbst aber keine XML-Dokumente, sondern folgen einer eigenen Syntax.

Eine DTD besteht aus einer Liste von Element- und Attributdefinitionen, deren Reihenfolge keine Rolle spielt. Elemente werden definiert mit

```
<!ELEMENT element content>
```

Inhalt von Elementen

Dieser Ausdruck legt fest, dass alle Elemente mit dem Namen *element* den Inhalt *content* haben. *content* kann einer der folgenden Ausdrücke sein:

`ANY`	Beliebiger Inhalt
`EMPTY`	Kein Inhalt, leeres Element
`(#PCDATA)`	Nur Text
`(#PCDATA | element | ...)*`	Mischung aus Text und Kindelementen in beliebiger Anzahl und Anordnung
`(children)`	Kindelemente in einer bestimmten Anordnung

[20] XHTML ist eine Variante von HTML. Im Gegensatz zu HTML verlangt XHTML Wohlgeformtheit im Sinne von XML. Die meisten Browser kommen mit XHTML und HTML gleichermaßen zurecht.

[21] Die Schreibweise „XML Schema“ mit zwei getrennten Wörtern ist zwar im Englischen korrekt, passt aber nicht in deutschen Texten. Der Bindestrich ist ein Kompromiss zwischen Lesbarkeit und Originaltreue.

Anordnung von Kindelementen

Die folgenden Schreibweisen für Kindelemente *children* legen bestimmte Anordnungen fest. Runde Klammern gruppieren Teilausdrücke und erlauben die Konstruktion geschachtelter Ausdrücke:

element	Ein Element mit dem Namen *element*
children , *children* , ...	Sequenz in fester Reihenfolge
children | *children* | ...	Auswahl aus der Liste
children?	Option
children+	Ein- oder mehrmalige Wiederholung
children∗	Beliebige Wiederholung, einschließlich Weglassen

Grenzen der Ausdrucksmächtigkeit

Die Schreibweise ist an reguläre Ausdrücke angelehnt. Sie ist eingängig, hat aber Grenzen. Probleme machen zum Beispiel „alle Elemente aus einer Auswahl in beliebiger Reihenfolge“ und „erst eine beliebige Anzahl Elemente x, dann gleich viele Elemente y“.

Die Definition einer Attributliste bezieht sich auf ein Element und nennt für jedes Attribut Namen, DTD-Typ und Vorgabewert. Die Reihenfolge der einzelnen Attributdefinitionen ist ohne Bedeutung.

```
<!ATTLIST element
   attributename type default
   ...
>
```

Eingrenzung von Attributwerten

Der DTD-Typ *type* begrenzt die möglichen Attributwerte:

`CDATA`	Beliebiger Text.
`ID`	Ein Bezeichner, den kein anderes Attribut vom DTD-Typ `ID` als Wert hat. Bezeichner unte
`IDREF`	Ein Bezeichner, der als Wert eines Attributs vom DTD-Typ `ID` vorkommt.
`IDREFS`	Mit Leerzeichen getrennte Liste von Bezeichnern, die als Werte von Attributen des DTD-
(*ident*|*ident*|...)	Einer der aufgelisteten Bezeichner.

Eindeutige und voreingestellte Attributwerte

Mit Attributen der DTD-Typen `ID`, `IDREF` und `IDREFS` können Elemente in verschiedenen Teilen des Dokuments aufeinander Bezug nehmen. Die Werte von `ID`-Attributen müssen eindeutig sein, unabhängig vom Attributnamen.[22] Der Vorgabewert *default* regelt den Umgang mit fehlenden Attributen.

`#REQUIRED`	Das Attribut ist verpflichtend, es darf nicht fehlen.
`#IMPLIED`	Optionales Attribut ohne Defaultwert.
"*text*"	Optionales Attribut mit dem Defaultwert *text*.

[22] IDs werden beispielsweise zur Serialisierung mit JavaBeans (Seite 153) und XStream (Seite 163) gebraucht.

Das folgende Beispiel zeigt eine DTD zum XML-Dokument `time.xml` (Listing 3.2). Diese DTD regelt, dass Stunden und Minuten genannt werden müssen, die Sekundenangabe aber optional ist. Auch die `time`-Attribute `zone` und `summer` sind optional. Wenn `summer` genannt wird, muss der Wert `true` sein.

```
<!ELEMENT time (hour, minute, second?)>
<!ELEMENT hour (#PCDATA)>
<!ELEMENT minute (#PCDATA)>
<!ELEMENT second (#PCDATA)>
<!ATTLIST time
    zone    CDATA           #IMPLIED
    summer  (true)          "true"
>
```

Listing 3.8: DTD für XML-Dokumente mit Zeitangaben.

Bezug zwischen XML-Dokument und DTD

Um ein XML-Dokument mit einer DTD zusammenzuführen, gibt es die folgenden Möglichkeiten.

- Das XML-Dokument nimmt selbst keinen Bezug auf die DTD, die in einer eigenständigen Textdatei gespeichert ist. Erst die Validierung verknüpft das XML-Dokument mit der DTD.
- Das XML-Dokument referenziert die DTD als externes Dokument wie im folgenden Beispiel `time-external-dtd.xml`. In der zweiten Zeile ist der Name des Wurzelelements und die Textdatei angegeben, in der die DTD gespeichert ist.[23]

```
<?xml version="1.0"?>
<!DOCTYPE time SYSTEM "time.dtd">
<time zone="GMT+1" summer="true">
   <hour>12</hour>
   <minute>50</minute>
</time>
```

Listing 3.9: XML-Dokument mit Referenz auf eine externe DTD.

Das XML-Dokument enthält eine interne DTD, wie im folgenden Beispiel `time-internal-dtd.xml`:

-
```
<?xml version="1.0"?>
<!DOCTYPE time [
    <!ELEMENT time (hour, minute, second?)>
    <!ELEMENT hour (#PCDATA)>
    <!ELEMENT minute (#PCDATA)>
    <!ELEMENT second (#PCDATA)>
    <!ATTLIST time
        zone    CDATA           #IMPLIED
```

[23] Allgemein ist hier eine URL zulässig, die die DTD zur Verfügung stellt.

```
        summer  (true)          "true"
    >
]>
<time zone="GMT+1" summer="true">
   <hour>12</hour>
   <minute>50</minute>
</time>
```

Listing 3.10: XML-Dokument mit einer internen (eingebetteten) DTD.

Validierung gegenüber einer DTD

Zur **Validierung** wird ein Werkzeug gebraucht, das XML-Dokumente und DTDs liest und abgleicht. Das frei verfügbare Kommandozeilenprogramm `xmllint` ist ein solches Werkzeug.[24]

Im folgenden Beispiel validiert `xmllint` das XML-Dokument `time-attribute.xml` (Listing 3.3) gegenüber der eigenständigen DTD-Datei `time.dtd` (Listing 3.8). Das Programm findet keine Fehler und gibt nichts aus, das Dokument ist also valide.[25]

```
$ xmllint --noout --dtdvalid time.dtd time-attribute.xml
$
```

Das folgende Dokument enthält ein überzähliges Element `foobar`:

```
<?xml version="1.0"?>
<time zone="GMT+1" summer="true">
   <foobar>Junk</foobar>
   <hour>12</hour>
   <minute>50</minute>
</time>
```

`xmllint` entdeckt und meldet den Fehler. Das Dokument `time-foobar.xml` ist nicht valide bezüglich `time.dtd` (Listing 3.8):

```
$ xmllint --noout --dtdvalid time.dtd time-foobar.xml
time-foobar.xml:2: element time: validity error :
    Element time content does not follow the DTD,
    expecting (hour , minute , second?), got (foobar hour minute )
time-foobar.xml:3: element foobar: validity error :
    No declaration for element foobar
Document time-foobar.xml does not validate against time.dtd
$
```

[24] `xmllint` ist Teil des „XML C Parser Toolkit“, das ursprünglich für den Unix-Desktop Gnome entwickelt wurde. Die Bibliothek ist aber auch ohne Gnome verwendbar. Fertige, ausführbare Versionen von `xmllint` stehen für alle verbreiteten Systeme zur Verfügung.

[25] Die meisten Programme aus dem Unix-Umfeld geben möglichst wenig aus. Keine Ausgabe bedeutet also, dass alles in Ordnung ist.

Schwächen der DTD

Allerdings klassifiziert `xmllint` auch das folgende XML-Dokument als valide, obwohl es offensichtlich keine brauchbare Zeitangabe enthält:[26]

```
<?xml version="1.0"?>
<time zone="Abrakadabra Simsalabim" summer="true">
    <hour/>
    <minute>-6666666666</minute>
</time>
```

Listing 3.11: Gemäß DTD valides XML-Dokument mit einer unsinnigen Zeitangabe.

Hier zeigt sich eine der Schwächen von DTDs. Abgesehen von der eigenwilligen Syntax[27] ist die Ausdrucksmächtigkeit begrenzt. So kann der Inhalt von Textknoten und Attributwerten kaum kontrolliert werden.[28] Weiter gibt es keine Möglichkeit, ähnliche Teilstrukturen zusammenzufassen. Die Unterstützung von Namespaces fehlt, ebenso wie ein Mechanismus um unabhängige DTDs für Teildokumente zu definieren und wieder zu kombinieren.

XML-Schema

XML-Schreibweise für XML-Grammatiken

Eine **XML-Schema-Definition** (XSD, kurz „XML-Schema“) hat den gleichen Zweck wie eine DTD, erlaubt aber eine genauere Kontrolle. Abgesehen davon ist eine XSD selbst ein XML-Dokument. XSDs sind damit zwar für Menschen schwerer zu lesen als DTDs, aber maschinell mit den gleichen Mitteln zu verarbeiten wie alle XML-Dokumente. Zum Beispiel definiert die Grammatik `http://www.w3.org/2001/XMLSchema` den Aufbau von XML-Schema.[29]

Der Rahmen einer XSD sieht folgendermaßen aus:

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
    ...
</schema>
```

Nachbildung der Elementstruktur

Unter dem Wurzelelement findet sich eine Art Abbild der Struktur des beschrie-

[26] Um Missverständnissen vorzubeugen: Das ist ein Problem der DTDs, nicht von `xmllint`.

[27] Diese Syntax ist Teil des älteren Formalismus „SGML“, aus dem auch XML entstanden ist. SGML hat wenig Verbreitung gefunden.

[28] Die einzige Ausnahme sind Attribute der DTD-Typen `ID`, `IDREF` und `IDREFS`, deren Attributwerte ausgewertet und abgeglichen werden.

[29] Diese Grammatik kann sich selbst validieren.

benen XML-Dokuments. Elemente der Grammatik werden zu Elementen namens `element`. Das Dokument

```
<?xml version="1.0"?>
<time>
    <hour>12</hour>
    <minute>50</minute>
</time>
```

wird definiert durch die XSD

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="time">
        ...
        <element name="hour">...</element>
        <element name="minute">...</element>
        ...
    </element>
</schema>
```

Die Elemente des Dokuments (`time`, `hour`, `minute`) werden im Schema in der gleichen Anordnung zu `element`-Elementen mit entsprechenden `name`-Attributen.

Schema-Typen für den Inhalt von Elementen

So genannte **Schema-Typen** beschreiben den Inhalt von Elementen. Es gibt einfache und komplexe Schema-Typen (*simple types*, *complex types*).

Einfache Schema-Typen

Elemente, die nur Text enthalten, haben einen einfachen Schema-Typ. XML-Schema gibt eine ganze Reihe einfacher Schema-Typen vor, mit denen der Aufbau des Textinhalts eingeschränkt werden kann. Die einfachen Schema-Typen sind an Typen von Programmiersprachen angelehnt. Beispiele sind:

`string`	Beliebiger Text
`integer`	Ganze Zahl
`nonNegativeInteger`	Nicht negative ganze Zahl
`boolean`	`true` oder `false`
`double`	Gleitkommazahl

Die Stunden- und Minutenzahl in der Zeitangabe lässt sich damit genauer festlegen:

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
```

```
    <element name="time">
        ...
        <element name="hour" type="nonNegativeInteger">...</element>
        <element name="minute" type="nonNegativeInteger">...</element>
        ...
    </element>
</schema>
```

Einschränkung des Wertebereichs

Die einfachen Schema-Typen können noch weiter beschränkt werden durch zusätzliche Angabe von

- Grenzwerten (Attribute `minInclusive` und `maxExclusive`) bei Zahlen,
- regulären Ausdrücken (Attribut `pattern`) bei Strings und
- Aufzählungen (Attribut `enumeration`) bei Strings.

Im folgenden Beispiel werden die Stunden- und die Minutenangabe auf sinnvolle Werte (0–23 und 0–59) begrenzt:

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="time">
        ...
        <element name="hour">
            <simpleType>
                <restriction base="nonNegativeInteger">
                    <maxExclusive value="24"/>
                </restriction>
            </simpleType>
        </element>
        <element name="minute">
            <simpleType>
                <restriction base="nonNegativeInteger">
                    <maxExclusive value="60"/>
                </restriction>
            </simpleType>
        </element>
        ...
    </element>
</schema>
```

Komplexe Schema-Typen für zusammengesetzte Strukturen

Elemente mit einem anderen Inhalt als Textknoten haben einen komplexen Schema-Typ. Das betrifft alle Elemente mit geschachtelten Elementen als Kindern. „Ordnungsindikatoren“ legen die Anordnung der Kindelemente fest:

`all`	Jedes der aufgezählten Elemente einmal in beliebiger Reihenfolge.
`sequence`	Jedes der aufgezählten Elemente in der angegebenen Reihenfolge.
`choice`	Eines der aufgezählten Elemente.

Damit kann das XML-Schema für Zeitangaben weiter vervollständigt werden.

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="time">
        <complexType>
            <sequence>
                <element name="hour">...</element>
                <element name="minute">...</element>
            </sequence>
        </complexType>
    </element>
</schema>
```

Anzahl Vorkommen von Kindelementen

Die Anzahl der Vorkommen kann mit den Attributen `minOccurs` und `maxOccurs` eingegrenzt werden, deren Werte vorzeichenlose ganze Zahlen sind.[30] Für `maxOccurs` ist außerdem die Angabe `unbounded` erlaubt, wenn die Anzahl Vorkommen beliebig ist. Die Voreinstellung bei fehlender Angabe ist 1.

Definition von Attributen

XSD-Elemente `attribute` definieren Attribute des Dokuments. Auch `attribute`-Elemente gehören zu Schema-Typen. Das folgenden Beispiel ergänzt den Schema-Typ von `time`-Elementen um die beiden Attribute `zone` und `summer`:

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="time">
        <complexType>
            <sequence>...</sequence>
            <attribute name="zone">...</attribute>
            <attribute name="summer">...</attribute>
        </complexType>
    </element>
</schema>
```

Für Attributwerte stehen die gleichen einfachen Schema-Typen (*simple type*) zur Verfügung wie für Textknoten.[31]

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="time">
```

[30] Beim Ordnungsindikator `all` ist `maxOccurs` auf den Wert 1 fixiert. Für `minOccurs` ist nur 0 oder 1 zulässig.

[31] Der reguläre Ausdruck für die Werte von `zone` definiert die textuelle Form von Zeitzonen, lässt aber immer noch sinnlose Angaben zu, wie zum Beispiel `XXX-99`. Dieser reguläre Ausdruck ist ein Kompromiss zwischen Aufwand und Korrektheit.

```
        <complexType>
            <sequence>...</sequence>
            <attribute name="zone">
                <simpleType>
                    <restriction base="string">
                        <pattern value="[A-Z]{3}[-+][0-9]{1,2}"/>
                    </restriction>
                </simpleType>
            </attribute>
            <attribute name="summer" type="boolean" default="true"/>
        </complexType>
    </element>
</schema>
```

Schema-Typen können mit Typnamen versehen und dann mehrfach referenziert und geschachtelt werden. Im folgenden Beispiel werden Zeitangaben um ein optionales Sekunden-Element ergänzt, dessen Inhalt den gleichen Typ wie das Minuten-Element hat. Um eine Wiederholung zu vermeiden, ist dieser Typ explizit als `Number60` definiert.

Daraus ergibt sich ein technisches Problem: Im Wurzelelement wurde bisher die Schema-Grammatik als Default-Namespace definiert (siehe Seite 184). Der selbst definierte Typ `Number60` existiert dort natürlich nicht. Ein Namespace-Präfix für XML-Schema und eine eindeutige Zuordnung der Element- und Attributnamen lösen das Problem:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="time" type="Time"/>
    <xsd:complexType name="Time">
        <xsd:sequence>
            <xsd:element name="hour">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:nonNegativeInteger">
                        <xsd:maxExclusive value="24"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:element>
            <xsd:element name="minute" type="Number60"/>
            <xsd:element name="second" type="Number60" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="zone">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:pattern value="[A-Z]{3}[-+][0-9]{1,2}"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="summer" type="xsd:boolean" default="true"/>
    </xsd:complexType>
    <xsd:simpleType name="Number60">
        <xsd:restriction base="xsd:nonNegativeInteger">
```

```
            <xsd:maxExclusive value="60"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>
```

Listing 3.12: `time.xsd`, XML-Schema für XML-Dokumente mit Zeitangaben.

Validierung gegenüber einem XML-Schema

Zur Validierung stehen viele Werkzeuge zur Verfügung. Das Kommandozeilenwerkzeug `xmllint` kann auch mit XSD umgehen, um beispielsweise `time-attribute.xml` (Listing 3.3) zu validieren:

```
$ xmllint --noout --schema time.xsd time-attribute.xml
time-attribute.xml validates
$
```

`xmllint` erkennt jetzt das XML-Dokument `time-funny.xml` (Listing 3.11) mit unsinnigem Inhalt, das mit einer DTD fälschlich als valide klassifiziert wurde, als unzulässig:

```
$ xmllint --noout --schema time.xsd time-funny.xml
time-funny.xml:2: element time: Schemas validity error :
    Element 'time', attribute 'zone': [facet 'pattern']
    The value 'Abrakadabra Simsalabim' is not accepted by the pattern '[A-Z]{3}[-+][0-9]{
time-funny.xml:2: element time: Schemas validity error :
    Element 'time', attribute 'zone':
    'Abrakadabra Simsalabim' is not a valid value of the local atomic type.
time-funny.xml:3: element hour: Schemas validity error :
    Element 'hour': '' is not a valid value of the local atomic type.
time-funny.xml:4: element minute: Schemas validity error :
    Element 'minute': '-6666666666' is not a valid value of the atomic type 'Number60'.
time-funny.xml fails to validate
$
```

Bezug zwischen XML-Dokument und -Schema

Wie bei DTDs gibt es verschiedene Möglichkeiten, um ein XML-Dokument mit einem XML-Schema in Verbindung zu bringen (siehe Seite 188):

- Das XML-Dokument weist keinen Bezug zu einem Schema aus. Das Schema wird erst bei der Validierung festgelegt. Nach diesem Verfahren gehen die bisher gezeigten Beispiele vor.
- Das XML-Dokument referenziert selbst ein bestimmtes Schema. Ein Paar von Attributen im Wurzelelement liefert die nötige Information:

`xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`
: Definiert das Präfix `xsi` für die nachfolgende Angabe eines XML-Schemas.

`xsi:noNamespaceSchemaLocation="`*filename*`"`
: Legt fest, dass die Datei *filename*[32] das XML-Schema für alle Elemente und Attribute ohne Namespace-Präfix beisteuert.

Das folgende XML-Dokument bezieht sich auf die Schema-Datei `time.xsd` (Listing 3.12):

```
<?xml version="1.0"?>
<time
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="time.xsd"
    zone="GMT+1"
    summer="true">
    <hour>12</hour>
    <minute>50</minute>
</time>
```

Listing 3.13: XML-Dokument mit Referenz auf ein XML-Schema.

Anders als bei DTDs wäre ein Dokument mit einem eingebetteten Schema nicht sehr sinnvoll. Das Schema müsste sich in diesem Fall als Teil des Dokuments selbst validieren.[33]

Java bietet selbst die Mittel zur Validierung, wie der nächste Abschnitt zeigt. Ein Java-Programm kann damit die Aufgabe von Werkzeugen wie `xmllint` übernehmen.

3.2 Arbeit mit dem DOM

Objektrepräsentation einer XML-Struktur

Ein „Document Object Model“ (DOM) ist eine programminterne Darstellung eines kompletten XML-Dokuments. Die verschiedenen Bestandteile des XML-Dokuments werden im DOM durch Java-Objekte repräsentiert. Die Objekte sind als Baum mit der gleichen Struktur wie das Dokument organisiert.[34]

[32] Der Attributwert kann eine beliebige URL nennen, unter der das Schema zu finden ist. Eine lokale Datei ist nur ein einfacher Sonderfall.

[33] DTDs mit ihrer eigenen Syntax mischen sich nicht mit den übrigen Elementen. Das Problem stellt sich dort daher nicht.

[34] Streng genommen gilt das nicht für Attribute eines Elements, die keiner bestimmten Anordnung unterliegen.

Ein **DOM-Parser** liest ein XML-Dokument und liefert die Baumstruktur. Der Baum kann dann durchsucht, verändert und schließlich wieder als XML-Dokument oder in einer anderen Form ausgegeben werden.

Merkmale eines DOM-Parsers

Diese Vorgehensweise holt das *komplette* Dokument in das Programm. Das hat Vor- und Nachteile. Der wesentliche Vorteil ist die Möglichkeit, beliebige Stellen im Baum gleichzeitig anzusprechen. Ein Nachteil ist die Begrenzung des Datenvolumens, das komplett in den Speicher passen muss. Eine Alternative mit entgegengesetzten Eigenschaften ist die sequenzielle Verarbeitung mit einem SAX-Parser (Seite 214).

3.2.1 DOM-Parser

DOM-Parser in der Java-Bibliothek

Einen XML-Parser zu schreiben ist keine einfache Aufgabe. Glücklicherweise gibt es eine ganze Reihe fertiger XML-Parser. Auch die Java-Bibliothek enthält einen „DOM-Parser“.[35]

DOM-Parser sind Klassen, die von der abstrakten Basisklasse `DocumentBuilder` im Package `javax.xml.parsers` abgeleitet sind. `DocumentBuilder` werden allerdings nicht mit Konstruktoren erzeugt, sondern mit der Factory-Methode `newDocumentBuilder` der Klasse `DocumentBuilderFactory`. Zunächst muss also eine `DocumentBuilderFactory` beschafft werden:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
```

Der zweite Schritt erzeugt mithilfe der Factory einen neuen DOM-Parser:

```
DocumentBuilder builder = factory.newDocumentBuilder();
```

Ein DOM-Parser definiert als wichtigste Methode `parse`, die ein XML-Dokument liest und die Wurzel des DOM liefert. Auf den Typ des Wurzelknotens, `Document`, geht der nächste Abschnitt ein.

```
Document document = builder.parse(...);
```

Aufruf eines DOM-Parsers

Das folgende Beispielprogramm zeigt den Ablauf:

[35] Der DOM-Parser in der Java-Bibliothek beruht auf dem Open-Source-Parser „Xerces“ der Apache Foundation.

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;

public class ParseDOM {
    public static void main(String... args) throws ParserConfigurationException, SAXExcep
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.parse(args[0]);
    }
}
```

Listing 3.14: Lesen eines XML-Dokumentes in ein DOM.

Es gibt auch andere Wege eine XML-Datei in ein DOM zu parsen. Das hier gezeigte Verfahren hat den Vorteil, dass es sich mit wenig Aufwand auf sequenzielle Verarbeitung (Seite 214) übertragen lässt.[36]

3.2.2 Validierung

Einbindung einer DTD zur Validierung

Der Standard-DOM-Parser von Java setzt ein wohlgeformtes XML-Dokument voraus, validiert das Dokument aber in der Voreinstellung nicht.

DTD

Validierung gegenüber einer DTD lässt sich in der `DocumentBuilderFactory` mit dem Setter `setValidating` anfordern. Die Factory produziert im folgenden Beispiel einen validierenden Parser:

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;

public class ParseDOMValidateDTD {
    public static void main(String... args) throws ParserConfigurationException, SAXExcep
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setValidating(true);
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.parse(args[0]);
    }
}
```

[36] Zum Beispiel spezifiziert DOM-Version 3 das Interface „Load and Save“, dessen Java-Implementierung im Package `org.w3c.dom.ls` zu finden ist. „Load and Save“ ist allerdings DOM-spezifisch.

Listing 3.15: Validieren eines XML-Dokumentes gegenüber einer externen oder internen DTD.

Dieses Programm validiert das XML-Dokument beim Lesen und benutzt dazu die im Dokument eingebettete oder referenzierte DTD. Ein Dokument ohne Angabe einer DTD kann nicht ohne Weiteres validiert werden.[37]

XML-Schema

Validierung mit einem XML-Schema

Der Setter `setValidating` aktiviert den Einsatz von DTDs, wie im vorhergehenden Beispielprogramm gezeigt. Um stattdessen gegenüber einem XML-Schema zu validieren, sind ein paar weitere Maßnahmen nötig.

Eine `DocumentBuilderFactory` kennt eine Reihe von „Factory-Attributen“[38], die ihre Arbeitsweise beeinflussen. Das hier entscheidende Factory-Attribut namens[39]

```
http://java.sun.com/xml/jaxp/properties/schemaLanguage
```

legt die Art der Grammatik fest, die der nachher produzierte XML-Parser zum Validieren verwendet. Der Attributwert `"http://www.w3.org/2001/XMLSchema"` wählt XML-Schema aus:

```
factory.setAttribute("http://java.sun.com/xml/jaxp/properties/schemaLanguage",
                     "http://www.w3.org/2001/XMLSchema");
```

Einbindung von Namespaces

XML-Schemata erfordern darüber hinaus Namespaces, die ein Parser in der Voreinstellung nicht berücksichtigt. Der Setter `setNamespaceAware` ändert das:

```
factory.setNamespaceAware(true);
```

Das folgende Programm validiert ein XML-Dokument gegenüber einem XML-Schema, das im Dokument selbst genannt ist (siehe Seite 196):

```
import java.io.*;
import javax.xml.parsers.*;
```

[37] Ein Umweg führt über sogenannte Transformationen (siehe auch Seite 209).

[38] Vorsicht: Diese Attribute haben nichts mit den Attributen in einem XML-Dokument zu tun!

[39] Der Name dieses Factory-Attributs hat die Gestalt einer URL. Das ist eine reine Äußerlichkeit. Die Factory behandelt diese URL nur als String und nimmt keinen Kontakt mit der Webseite auf. Entsprechendes gilt für den Wert des Factory-Attributs, der ebenfalls wie eine URL geschrieben wird.

```
import org.w3c.dom.*;
import org.xml.sax.*;

public class ParseDOMValidateXSD {
    public static void main(String... args) throws ParserConfigurationException, SAXExcep
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setValidating(true);
        factory.setNamespaceAware(true);
        factory.setAttribute("http://java.sun.com/xml/jaxp/properties/schemaLanguage",
                             "http://www.w3.org/2001/XMLSchema");
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.parse(args[0]);
    }
}
```

Listing 3.16: Validieren eines XML-Dokumentes mit einem XML-Schema, das im Dokument genannt ist.

Externes Schema gegenüber einem internen Schema

Ähnlich einfach lässt sich ein beliebiges anderes, vom Dokument unabhängiges XML-Schema verwenden. Der Wert des weiteren Factory-Attributs

```
http://java.sun.com/xml/jaxp/properties/schemaSource
```

legt das gewünschte Schema fest. Schiebt man in `ParseDOMValidateXSD` (Listing 3.16) noch die Anweisung

```
factory.setAttribute("http://java.sun.com/xml/jaxp/properties/schemaSource",
                     args[1]);
```

ein, dann erwartet das Programm als zweites Kommandozeilenargument den Namen einer XML-Schema-Datei[40] und verwendet dieses Schema zur Validierung.

Ein auf diese Art explizit festgelegtes Schema hat in jedem Fall Vorrang. Eine Angabe innerhalb des Dokuments, ob DTD oder Schema, wird ignoriert. Dieses Verhalten ist durchaus sinnvoll: Einem selbst ausgewählten Schema bringt man möglicherweise mehr Vertrauen entgegen als einer Grammatik, die sich das Dokument selbst wünscht.

Ein anderer Weg zur Validierung führt über Klassen im Package `javax.xml.validation`. Dieses Verfahren trennt die Validierung vom Parsen und trägt zur Modularisierung bei. Es eignet sich allerdings nicht für DTDs und wird hier zugunsten einer einfacheren Darstellung beiseite gelassen.

[40] Genau genommen kann der Wert eine beliebige URL sein, unter der das Schema zur Verfügung steht. Es wird gegebenenfalls heruntergeladen.

3.2.3 Objektbaum

Knotenarten in einem Objektbaum

Ein DOM-Parser liefert das DOM als Baum von Java-Objekten. Die verschiedenen Bestandteile dieses Baums sind durch unterschiedliche Knotenarten repräsentiert. Beispiele sind Elementknoten und Textknoten. Die Auswahl der Knotenarten ist unabhängig von Java und gilt für alle Programmiersprachen. Die entsprechenden Java-Typen sind in dem dedizierten Package `org.w3c.dom` definiert und folgen nicht ganz den in Java üblichen Regeln. Der Umgang mit diesen Klassen und Typen wirkt aus dem Blickwinkel von Java etwas ungewohnt und sperrig.

Interface `Node`

Gemeinsames Interface für Knotenklassen

Alle Knoten des DOM sind kompatibel zum Interface `Node`. Das gilt zum Beispiel auch für das Interface `Document`, das die Wurzel des DOM repräsentiert. Die XML-Parser liefern ein `Document`-Objekt als Ergebnis.

Die folgenden Methoden geben Auskunft über einen Knoten des DOM:

`short getNodeType()`
: Liefert die Art des Knotens als Konstante.

`String getNodeName()`
: Liefert den Namen des Knotens. Die Interpretation des Namens hängt vom Knotentyp ab.

`String getNodeValue()`
: Liefert den Wert des Knotens. Auch die Interpretation des Werts hängt vom Knotentyp ab.

`NodeList getChildNodes()`
: Liefert die direkten Kindknoten.

Konstanten für Knotenarten

Im Interface `Node` sind Konstanten für die verschiedenen Knotenarten definiert:[41]

Knotentyp	Knotenart	Name	Wert
`ELEMENT_NODE`	Element	Elementname	`null`
`DOCUMENT_NODE`	Gesamtes Dokument samt XML-Deklaration	`"#document"`	`null`
`TEXT_NODE`	Textknoten, nicht markiert	`"#text"`	Text
`CDATA_SECTION_NODE`	CDATA-Abschnitt	`"#cdata-section"`	Text
`ATTRIBUTE_NODE`	Attribut	Attributname	Attributwert

[41] Die Konstanten in der linken Spalte sind öffentliche Klassenvariablen vom Typ `short`. Für Java ist das eine eher ungewöhnliche Wahl.

Dazu sind ein paar Anmerkungen nötig:

- Das Wurzelelement, das `parse` direkt liefert, ist das einzige mit dem Knotentyp `DOCUMENT_NODE`.
- Text taucht in Knoten zweier verschiedene Typen auf, `TEXT_NODE` und `CDATA_SECTION_NODE`. Für den Zugriff auf den Inhalt spielt das keine Rolle.
- Attributknoten hängen im Baum zwar am attributierten Element, werden aber anders als Kindknoten angesprochen.

Namen und Werte von Knoten

Technisch hat jeder Knoten einen Namen und einen Wert. Abhängig vom Knotentyp ist der Wert aber manchmal nur ein Platzhalter ohne sinnvolle Information, wie den beiden rechten Spalten der Tabelle zu entnehmen ist.[42]

Navigation im DOM

Zugriff auf die Baumstruktur

Die Baumstruktur des DOM erschließt der Getter `getChildNodes`, der die Kindknoten verpackt in eine `NodeList` liefert. [43] Der Inhalt einer `NodeList` lässt sich mit den beiden folgenden Methoden auslesen:

`int getLength()`
: Liefert die Anzahl der Knoten in dieser Liste.

`Node item(int index)`
: Liefert den Knoten am gegebenen Index. Bei einem ungültigen Index ist das Ergebnis `null`.[44]

Beispiel für den Durchlauf eines Baums

Das folgende Beispielprogramm liest die XML-Datei. Es durchläuft mit der Methode `walk` das DOM und gibt für jeden erreichten Knoten eine Zeile aus. Der zweite Parameter von `walk`, `indent`, dient nur zum Einrücken der Ausgabezeilen und hat keinen Einfluss auf die Funktionsweise:

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
```

[42] Dieses Verhalten ist ein Zugeständnis der Java-Implementierung an die Vorgaben des Standards.

[43] Diese anscheinend einfache Klasse ist nicht nur ein Array von Knoten. Änderungen am DOM spiegeln sich „live“ in `NodeList`-Objekten wider. Die Klasse bietet damit eine „Sicht“ auf einen Ausschnitt des DOM.

[44] Hier zeigt sich einmal mehr der Einfluss des Standards. Java reagiert im Allgemeinen mit Exceptions auf solche Indexfehler.

```
import org.xml.sax.*;

public class WalkDOM {
    public static void main(String... args) throws SAXException, ParserConfigurationExcep
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.parse(args[0]);
        walk(document, "");
    }

    static void walk(Node node, String indent) {
        System.out.printf(indent + "type=%d, name=%s, value=[%s]%n",
                          node.getNodeType(), node.getNodeName(), node.getNodeValue());
        NodeList children = node.getChildNodes();
        for(int i = 0; i < children.getLength(); i++)
            walk(children.item(i), indent + ":   ");
    }
}
```

Listing 3.17: Durchlauf aller Knoten eines DOM.

Startet man `WalkDOM` mit der Datei `time-attribute.xml` (Listing 3.3) dann erhält man die folgende Ausgabe.[45]

```
$ java WalkDOM time.xml
type=9, name=#document, value=[null]
:   type=1, name=time, value=[null]
:   :   type=3, name=#text, value=[
    ]
:   :   type=1, name=hour, value=[null]
:   :   :   type=3, name=#text, value=[12]
:   :   type=3, name=#text, value=[
    ]
:   :   type=1, name=minute, value=[null]
:   :   :   type=3, name=#text, value=[50]
:   :   type=3, name=#text, value=[
]
$
```

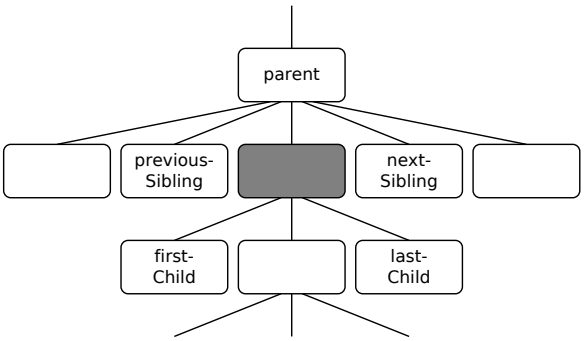
9, 1 und 3 sind die Werte der Konstanten `DOCUMENT_NODE`, `ELEMENT_NODE` und `TEXT_NODE`.

Ausgehend von einem Bezugsknoten navigieren die folgenden Methoden zu benachbarten Knoten im DOM. Das Ergebnis ist `null`, wenn kein entsprechender Knoten existiert.

```
Node  getFirstChild()
Node  getLastChild()
Node  getPreviousSibling()
Node  getNextSibling()
Node  getParentNode()
```

[45] Hier fehlen noch die Attributknoten. Das wird unten behoben.

Die Namen erklären sich mit der folgenden Skizze:



Attribute

Zugriff auf Attribute

In der Ausgabe des Beispielprogramms `WalkDOM` fehlen die Attribute. Attributknoten erfordern eine Sonderbehandlung: Statt `getChildNodes` liefert der Getter `getAttributes` die Attributknoten.[46] Das Ergebnis von `getAttributes` ist ein Objekt vom Typ `NamedNodeMap`. Eine `NamedNodeMap` hat die gleichen Methoden wie eine `NodeList`, legt aber keine bestimmte Reihenfolge der Elemente fest.[47]

Eine Erweiterung des Programms `WalkDOM` (Listing 3.17) kann auch Knoten mit dem Typcode 2, dem Wert der Konstanten `ATTRIBUTE_NODE`, ausgeben. Dazu ist das folgende Codefragment nach der `printf`-Anweisung in der Methode `walk` nötig:

```
NamedNodeMap attributes = node.getAttributes();
if(attributes != null)
    for(int i = 0; i < attributes.getLength(); i++)
        walk(attributes.item(i), indent + ":   ");
```

Listing 3.18: Besuch der Attributknoten in einem DOM.

Ein Aufruf zeigt auch die Attribute:

```
$ java WalkDOM time.xml
type=9, name=#document, value=[null]
:   type=1, name=time, value=[null]
:   :   type=2, name=summer, value=[true]
```

[46] `getAttributes` liefert bei Knoten anderer Typen als `ELEMENT_NODE` immer `null`.

[47] Die Reihenfolge kann sich bei jeder Modifikation des DOM ändern. Verschiedene Parser-Implementierungen können verschiedene Reihenfolgen liefern.

```
:   :   :   type=3, name=#text, value=[true]
:   :   type=2, name=zone, value=[GMT+1]
:   :   :   type=3, name=#text, value=[GMT+1]
:   :   type=3, name=#text, value=[
    ]
:   :   type=1, name=hour, value=[null]
:   :   :   type=3, name=#text, value=[12]
:   :   type=3, name=#text, value=[
    ]
:   :   type=1, name=minute, value=[null]
:   :   :   type=3, name=#text, value=[50]
:   :   type=3, name=#text, value=[
]
```

Der Attributwert taucht zweimal auf: einmal als „Wert“ des Attributknotens selbst und außerdem in einem Textknoten, dem einzigen Kind von Attributknoten.

Vereinfachtes DOM

Die Ausgabe von `WalkDOM` enthält Textknoten mit Layout (Zeilenumbrüche, Einrückung und dergleichen). Dieses Layout mag im Einzelfall signifikant sein oder auch nicht. Eine passende Konfiguration der Factory führt zu einem Parser, der Layout verwirft und nicht in das DOM übernimmt. Das setzt allerdings eine DTD oder ein XML-Schema voraus, sonst könnte der Parser signifikanten und nicht signifikanten Zwischenraum nicht unterscheiden.[48] Die folgenden Aufrufe unterdrücken Textknoten, die nur Layout enthalten.[49]

Umgang mit Zwischenraum

Textknoten mit Layout

```
factory.setIgnoringElementContentWhitespace(true);
factory.setValidating(true);
```

Listing 3.19: Auslassen von Textknoten ohne druckbaren Inhalt beim Durchlauf eines DOM.

Ein Programmstart mit der XML-Datei `time-internal-dtd.xml` (Listing 3.10) mit eingebetteter DTD zeigt nur noch die „interessanten“ Knoten:[50]

```
$ java WalkDOM time-internal-dtd.xml
type=9, name=#document, value=[null]
```

[48] Der Parser muss dazu nicht validieren.

[49] Das XML-Dokument muss in diesem Beispiel eine DTD enthalten oder referenzieren.

[50] Der Parser übergeht nur Textknoten, die *ausschließlich* Zwischenraum enthalten. Alle anderen Textknoten bleiben vollständig erhalten, einschließlich führender, schließender und innerer Zwischenraumzeichen.

```
:   type=10, name=time, value=[null]
:   type=1, name=time, value=[null]
:   :   type=2, name=summer, value=[true]
:   :   :   type=3, name=#text, value=[true]
:   :   type=2, name=zone, value=[GMT+1]
:   :   :   type=3, name=#text, value=[GMT+1]
:   :   type=1, name=hour, value=[null]
:   :   :   type=3, name=#text, value=[12]
:   :   type=1, name=minute, value=[null]
:   :   :   type=3, name=#text, value=[50]
```

In dieser Ausgabe taucht ein neuer Knoten mit dem Code 10 auf. Dahinter verbirgt sich die eingebettete DTD.[51]

XML-Kommentare im DOM

Ähnliches gilt für XML-Kommentare. In der Voreinstellung überträgt ein Parser die Kommentare in Knoten vom Typ `COMMENT_NODE` in das DOM. Die meisten Anwendungen brauchen XML-Kommentare aber nicht. Der Aufruf

```
factory.setIgnoringComments(true);
```

weist die Factory an, einen Parser zu erzeugen, der Kommentare ignoriert.

Vereinfachung von Textknoten

Schließlich ist die Unterscheidung zwischen Textknoten mit und ohne `CDATA`-Entwertung zwar Voreinstellung, aber für Anwendungen oft nebensächlich. Die Methode `setCoalescing` vereinheitlicht die Textknoten. Der Methodenaufruf

```
factory.setCoalescing(true);
```

ergibt einen Parser, der den Knotentyp `CDATA_SECTION_NODE` durch `TEXT_NODE` ersetzt. Dabei könnten direkt benachbarte Textknoten entstehen, die ein solcher Parser auch gleich zu einem einzigen Textknoten verschmilzt.

Diese Maßnahmen führen zu einfacheren, kleineren DOMs, die allerdings formal nicht mehr alle Informationen des ursprünglichen XML-Dokuments enthalten.

3.2.4 Arbeiten mit dem DOM

Interfaces für einzelne Knotenarten

Die Programme im vorhergehenden Abschnitt durchlaufen das DOM rekursiv und besuchen dabei jeden Knoten. Prinzipiell lässt sich damit jede Information im Baum

[51] Der „Name“ dieses Knotens vom Typ `DOCUMENT_TYPE_NODE` ist der Name des Wurzelelements, `time`, der aus der DTD alleine nicht hervorgeht.

finden. Das Interface `Node` stellt dazu aber nur „generische“ Methoden zur Verfügung, die bei manchen Knotentypen interessante Informationen liefern, bei anderen nicht. Zum Beispiel haben Elementknoten immer den „Wert“ `null` und Textknoten immer den „Namen“ `#text`.

Im Package `org.w3c.dom` gibt es eine Reihe von spezifischen Interfaces für die einzelnen Knotenarten, die besser passende Methoden definieren. Diese Interfaces sind von `Node` abgeleitet:

`Element`	`ELEMENT_NODE`
`Document`	`DOCUMENT_NODE`
`Text`	`TEXT_NODE`
`CDATASection`	`CDATA_SECTION_NODE`
`Attr`	`ATTRIBUTE_NODE`

Nützliche Getter dieser Interfaces sind:

Vereinfachte, knotenspezifische Zugriffsmethoden

`boolean hasAttribute(String name) // Element`
: gibt Auskunft, ob das Element ein Attribut namens `name` hat oder nicht.

`String getAttribute(String name) // Element`
: liefert den Wert eines Attributs mit dem Namen `name`. Wenn das Attribut nicht existiert, wird ein Leerstring zurückgegeben.

`NodeList getElementsByTagName(String name) // Element, Document`
: liefert eine Liste aller Elemente mit dem Namen `name`.[52] Diese Methode sammelt alle passenden Knoten im ganzen Teilbaum ein. Das Ergebnis kann eine leere Liste sein, aber nicht `null`.

`String getTextContent()` // alle Interfaces
: liefert den konkatenierten Inhalt aller Textknoten im Teilbaum. Wenn es keine Textknoten gibt, wird ein Leerstring zurückgegeben.

Diese Methoden erlauben in vielen Fällen einfacheren Code und ersparen rekursive Aufrufe. Das folgende Programm erwartet zwei Kommandozeilenargumente, eine XML-Datei und einen Elementnamen. Es sucht alle Knoten mit dem gegebenen Namen und gibt deren gesammelten Textinhalt aus.

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
```

[52] Der Joker * passt zu allen Elementen.

```
public class FindElementsText {
    public static void main(String... args) throws ParserConfigurationException, SAXExcep
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.parse(args[0]);

        NodeList nodes = document.getElementsByTagName(args[1]);
        System.out.printf("found %d element(s)%n", nodes.getLength());

        for(int i = 0; i < nodes.getLength(); i++)
            System.out.println(nodes.item(i).getTextContent());
    }
}
```

Listing 3.20: Suche nach Elementen im DOM nach ihrem Namen und Ausgabe ihres Textinhaltes.

Ein Test zeigt das erwartete Ergebnis:

```
$ java FindElementsText time.xml hour
found 1 element(s)
12
```

Beispiel mit realistischen Daten

Interessanter ist der Versuch mit einem realistischen XML-Dokument. Google bietet zum Beispiel aggregierte Nachrichten als „RSS-Feeds“ an, die technische XML-Dokumente mit einer bestimmten XML-Grammatik sind. Mit dem Open-Source-Programm `wget`[53] kann man eine solche Datei holen und in einer lokalen XML-Datei abspeichern:[54]

```
$ wget --quiet --output-document=news.xml http://news.google.de/news\?output=rss
```

Durchsucht man diese Datei mit dem Programm `FindElementsText` nach `title`-Elementen, dann erhält man die Schlagzeilen des Tages:

```
$ java FindElementsText news.xml title
found 12 element(s)
Google News: Schlagzeilen
Google News: Schlagzeilen
Aktienkurse brechen ein: Sorgenkind Griechenland: ...
Supernova-Forschung Physik-Nobelpreis für drei Astronomen ...
...
```

[53] Dazu kann auch ein Browser verwendet werden.

[54] Der Backslash ist in diesem Beispiel nötig, um das Fragezeichen an der Unix-Shell vorbeizumogeln. Er gehört nicht zur URL.

Laden von XML-Dokumenten aus dem Internet

Die XML-Parser akzeptieren als Quellen nicht nur Dateinamen, sondern beliebige URLs. Damit kann man sich den Weg über eine lokale Datei sparen. Der folgende Aufruf holt das Dokument aus dem Internet und durchsucht es.[55] Das Ergebnis ist das gleiche wie vorher:

```
$ java FindElementsText http://news.google.de/news\?output=rss title
found 12 element(s)
...
```

3.2.5 Ausgabe

DOM als XML-Dokument ausgeben

Die Ausgabe eines DOM in eine XML-Datei ist aufwendiger als das Einlesen. Auch hier gibt es verschiedene Möglichkeiten. Ein Weg führt über sogenannte **Transformationen**, deren Zweck eigentlich der skriptgesteuerte Umbau von DOMs ist.[56] Das Ergebnis einer solchen Transformation kann an verschiedene Abnehmer weitergeleitet werden, unter anderem auch an eine Datei. Wendet man dabei nur eine „identische", leere Transformation an, so wird das DOM in die Datei geschrieben.

Umweg über eine leere Transformation

Zur Ausgabe sind drei Objekte nötig:

1. ein `Transformer`-Objekt, das die (in diesem Fall leere) Transformation repräsentiert,
2. ein `DOMSource`-Objekt als Eingabe für die Transformation und
3. ein `StreamResult`-Objekt, das das Ergebnis der Transformation aufnimmt.

Das `Transformer`-Objekt entsteht auf dem inzwischen bekannten Weg über eine Factory. Die so erzeugte Transformation ist leer und reicht für das hier verfolgte Ziel aus:

```
TransformerFactory factory = TransformerFactory.newInstance();
Transformer transformer = factory.newTransformer();
```

Ein als `DOMSource` verpackter `Document`-Knoten dient als Eingabe:

[55] Ohne Verbindung zum Internet wird eine `SocketException` geworfen.

[56] „Extensible Stylesheet Language Transformations" (XSLT) sind der prädestinierte Formalismus zur Transformation von XML-Dokumenten. XSLT ist eine eigenständige, von Java unabhängige Programmiersprache. Anders als der Name vermuten lässt, hat XSLT nichts mit dem „Styling" von Webseiten im Sinne von CSS (*Cascading Style Sheet*) zu tun.

```
DOMSource source = new DOMSource(document);
```

Als Transformationsziel lassen sich verschiedene Repräsentanten der Ausgabedatei verwenden, darunter ein `FileWriter` (Seite 77), ein `FileOutputStream` (Seite 39) und ein `File`-Objekt, wie im folgenden Beispiel:

```
StreamResult result = new StreamResult(new File(filename));
```

Schließlich löst ein Aufruf der Methode `transform` die eigentliche Transformation aus, die zwar nichts am DOM ändert, aber quasi „nebenbei“ die gewünschte Ergebnisdatei produziert:

```
transformer.transform(source, result);
```

Das folgenden Programm fasst den Ablauf zusammen. Es erwartet zwei Dateinamen auf der Kommandozeile. `CopyDOM` parst ein DOM von der ersten Datei und schreibt es in die zweite:

```
import java.io.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.w3c.dom.*;
import org.xml.sax.*;

public class CopyDOM {
    public static void main(String... args) throws ParserConfigurationException, SAXExcep
        // DOM lesen
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.parse(args[0]);

        // DOM schreiben
        TransformerFactory transformerFactory = TransformerFactory.newInstance();
        Transformer transformer = transformerFactory.newTransformer();
        DOMSource source = new DOMSource(document);
        StreamResult result = new StreamResult(new File(args[1]));
        transformer.transform(source, result);
    }
}
```

Listing 3.21: Lesen und Schreiben eines DOM.

3.2.6 Modifikation

Neben den Gettern gibt es auch modifizierende Methoden. Sie verändern ein existierendes DOM oder erzeugen ein komplett neues.

Verändern und Erzeugen eines DOM

Neue Knoten

Alle Knoten kennen „ihr“ `Document`-Objekt. Es gibt also keine „heimatlosen“ Knoten, die keinem DOM zugeordnet sind. Neue Knoten lassen sich nur aus einem `Document`-Objekt gewinnen und gehören dann automatisch zu eben diesem Dokument. Die folgenden `Document`-Methoden produzieren neue Knoten:

`Document`-Objekt als Bezugspunkt

```
Element  createElement(String name)
Text  createTextNode(String text)
CDATASection  createCDATASection(String text)
```

Wenn man aus dem Nichts ein komplett neues DOM aufbauen will, braucht man zuerst ein `Document`-Objekt. Dazu eignet sich ein `DocumentBuilder`, der nicht nur ein XML-Dokument parsen, sondern mit der Methode `newDoument` auch ein ganz neues DOM anlegen kann:

```
Document document = builder.newDocument();
```

Dieses DOM ist noch leer und enthält überhaupt keine Knoten.[57]

Knoten einbauen

Neu erzeugte Knoten kennen zwar bereits „ihr“ DOM, stehen aber noch außerhalb des Baums. Die folgende Methode fügt den Knoten `node` als Kind an:

Einfügen neuer Knoten in das DOM

```
Node  appendChild(Node node)
```

Das folgende Programm baut ein neues DOM von unten nach oben auf, also von den Blättern zur Wurzel.

```
import java.io.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
```

[57] Die Ausgabe liefert lediglich eine XML-Deklaration.

```
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.w3c.dom.*;

public class CreateDOM {
    public static void main(String... args) throws ParserConfigurationException, IOExcept
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.newDocument();

        Element minute = document.createElement("minute");
        minute.appendChild(document.createTextNode("50"));

        Element hour = document.createElement("hour");
        hour.appendChild(document.createTextNode("12"));

        Element time = document.createElement("time");
        time.appendChild(hour);
        time.appendChild(minute);

        document.appendChild(time);

        // DOM schreiben ...
    }
}
```

Listing 3.22: Aufbau eines neuen DOM.

Das Ergebnis ist eine neue XML-Datei, der jedes Layout fehlt:

```
$ java CreateDOM new.xml
$ cat new.xml
<?xml version="1.0" encoding="UTF-8"?><time><hour>12</hour><minute>50</minute></time>$
```

Lesbar formatierte Ausgabe

Die zur Ausgabe verwendete leere Transformation kann mit dem Aufruf

```
transformer.setOutputProperty(OutputKeys.INDENT, "yes");
```

angewiesen werden, die Ausgabe mit zusätzlichem Layout gut lesbar umzubrechen. Dabei entstehen allerdings neue Textknoten mit Zwischenraum. Streng genommen stimmt die Ausgabe nicht mehr mit dem ursprünglichen DOM überein.

```
$ java CreateDOM new.xml
$ cat new.xml
<?xml version="1.0" encoding="UTF-8"?>
<time>
   <hour>12</hour>
   <minute>50</minute>
</time>$
```

Manipulation der Baumstruktur

Einfügen, Ersetzen und Entfernen von Knoten

`appendChild` fügt einen neuen Kindknoten hinten an die bereits vorhandenen Kindknoten an. Die folgenden Methoden erlauben weitere lokale Änderungen der Kindknoten:

`Node removeChild(Node node)`
: Entfernt den Kindknoten `node` und liefert ihn zurück.

`Node replaceChild(Node node, Node old)`
: Ersetzt den Kindknoten `old` durch den Knoten `node` und liefert den alten Knoten zurück.

`Node insertBefore(Node node, Node ref)`
: Schiebt den Knoten `node` vor dem Kindknoten `ref` ein.

Text und Attribute ändern

Austausch von Text und Attributen

Die beiden nächsten Methoden ändern nicht notwendigerweise die Baumstruktur, sondern tauschen Text aus.

`void setNodeValue(String text)`
: Ersetzt den Wert des Knotens durch neuen Text. Die Bedeutung des Werts hängt vom Knotentyp ab, wie der Tabelle auf Seite 201 zu entnehmen ist.

`void setTextContent(String text)`
: Ersetzt alle Kindknoten durch einen einzigen neuen Textknoten mit dem Inhalt `text`. Diese Methode ist etwas gefährlich, weil sie ganze Teilbäume „abhängen“ kann.

Nur Elementknoten haben Attribute. Die beiden folgenden Methoden sind daher nur für `Element`-Objekte definiert. Sie manipulieren die Attributliste:

`void setAttribute(String name, String value)`
: Fügt dem Knoten ein neues Attribut `name` mit dem Wert `value` zu oder ersetzt den Wert, wenn es schon ein Attribut mit diesem Namen gibt. Die folgende Erweiterung von `CreateDOM` (Listing 3.22) fügt zwei Attribute ein:

```
time.setAttribute("zone", "GMT+1");
time.setAttribute("summer", "true");
```

Listing 3.23: Einbau von Attributen in ein DOM.

`void removeAttribute(String name)`
: Entfernt das Attribut `name`.

Hilfsmethoden

Flache und tiefe Kopie eines Teilbaums

Schließlich seien zwei Methoden erwähnt, die zwar keine neue Funktionalität beisteuern, aber Arbeit einsparen können:

`Node cloneNode(boolean deep)`
: Erzeugt mit dem Argument `true` einen tiefe Kopie des Knotens. Dabei werden alle direkten und indirekten Kindknoten bis hinab zu den Blättern kopiert. Das Ergebnis ist ein neuer Baum, der unabhängig vom Original ist.
Das Argument `false` führt zu einer flachen Kopie. Das Ergebnis ist ein kopierter Knoten, der dieselben Kindknoten referenziert wie das Original. Änderungen an den Kindknoten wirken sich also in Original und Kopie aus.

`void normalize()`
: Kürzen unnötiger Knoten
Diese Methode fasst alle Textknoten im Baum so weit wie möglich zusammen. Leere Textknoten verschwinden dabei ganz aus dem DOM. Nach dem Aufruf gibt es keine leeren und keine direkt benachbarten Textknoten mehr.

3.2.7 Fehlerbehandlung

Fluchtwerte statt Exceptions als Fehlersignal

Viele der Methoden im Package `org.w3c.dom` gehen etwas anders mit Fehlern um, als sonst in Java üblich. Probleme äußern sich oft in Fluchtwerten als Ergebnis und nicht als Exceptions. Darin drückt sich die generelle Maßgabe der DOM-Spezifikation aus, die Verarbeitung erst dann abzubrechen, wenn definitiv nicht mehr fortgefahren werden kann.[58] Beispielsweise ignorieren Knoten, bei denen ein Wert sinnlos ist (siehe Tabelle Seite 201), Aufrufe der Methode `setNodeValue` ohne jeden Protest. Der Aufrufer erhält keinerlei Hinweis darauf, dass sein Versuch ins Leere ging.

3.3 SAX-Parser

Verarbeitung beim Lesen

Die Arbeit mit einem DOM entspricht mehr dem klassischen Ablauf: Der DOM-

[58] Ob dieses Verhalten hilfreich ist oder nicht, steht auf einem anderen Blatt.

Parser liest zuerst ein komplettes Dokument in ein DOM, das der Rest des Programms anschließend weiterverarbeitet. SAX (*Simple API for XML*) verzahnt dagegen Parsen und Verarbeitung zeitlich. Immer wenn der Parser ein Stück XML[59] erkannt hat, wird die Anwendung benachrichtigt und kann entscheiden, wie weiter verfahren werden soll. In gewissem Sinn ähnelt SAX den I/O-Streams (Kapitel 1), die einen langen Datenstrom ebenfalls in kleinen Einheiten sequenziell abarbeiten.

Vor- und Nachteile

Bei SAX entscheidet die Anwendung, was mit den gelesenen XML-Fragmenten geschieht. Das hat eine Reihe von Vorteilen:

- Uninteressante Informationen können ignoriert werden.
- Die Anwendung kann die Verarbeitung beenden, wenn der Rest des Dokuments keine Rolle mehr spielt.
- Die Länge des Dokuments spielt keine Rolle.
- Die Bearbeitung eines Dokuments kann schon beginnen, bevor es überhaupt komplett existiert.

Dem stehen Nachteile gegenüber:

- Die Anwendung muss die hierarchische Struktur gegebenenfalls selbst rekonstruieren. Der Parser präsentiert einen „flachen“ Strom von Einzelteilen.
- Es gibt immer noch unbekannte, erst später erreichbare Teile des Dokuments.
- Umformungen, die entfernte Teile des Dokuments einbeziehen, sind schwierig.
- Die Anwendung sieht immer nur einen Punkt im Dokument. Sie kann weder zu früheren Teilen des Dokuments zurückkehren, noch einen unverbindlichen Blick auf vorausliegende Abschnitte werfen.

Geeignet zum effizienten Lesen

SAX eignet sich in erster Linie zum Parsen eines XML-Dokuments, aber weniger zum Ändern oder Schreiben. Zum Ausgleich sind SAX-Parser die schnellste und sparsamste Möglichkeit, um XML zu verarbeiten.

3.3.1 Initialisierung

Parser-Factory liefert Parser

Die Initialisierung eines SAX-Parsers beginnt mit einer `SAXParserFactory`, die selbst von einer statischen Factory-Methode geliefert wird:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
```

[59] In der Sprache des Compilerbaus handelt es sich dabei um Token.

Die `SAXParserFactory` produziert einen SAX-Parser mit voreingestellten Eigenschaften. Zum Beispiel validiert der Parser nicht und kennt keine Namespaces.

```
SAXParser parser = factory.newSAXParser();
```

Die wichtigste Methode des Parsers ist die mehrfach überladene Methoden `parse`. Anders als ein DOM-Parser liefert sie kein Ergebnis, sondern erwartet ein zweites Argument, auf das weiter unten eingegangen wird.

```
parser.parse(filename, new DefaultHandler());
```

Das folgende Programm fasst diese Schritte zusammen. Es erhält den Namen einer XML-Datei als Argument und parst diese Datei:

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class ParseSAX {
    public static void main(String... args) throws ParserConfigurationException, SAXExcep
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser parser = factory.newSAXParser();
        parser.parse(args[0], new DefaultHandler());
    }
}
```

Listing 3.24: SAX-Parser für ein XML-Dokument.

3.3.2 Handler

Aufruf von Methoden aus dem Parser heraus

SAX-Parser halten die Anwendung über den Fortschritt auf dem Laufenden, indem sie sogenannte **Handlermethoden** aufrufen. Was in diesen Methoden geschieht, spielt für den Parserablauf keine Rolle. Vier Interfaces definieren die Schnittstellen dieser Handlermethoden. Im zunächst wichtigsten Interface `ContentHandler` stecken die Methoden, an die der Parser die erkannten XML-Fragmente weitergibt.[60] Im Einzelnen sind das die folgenden Methoden:

```
void  startDocument()
void  endDocument()
void  startElement(String uri, String localName, String qName, Attributs atts)
```

[60] Das weitere Interface `ErrorHandler` definiert beispielsweise Methoden für den Fehlerfall (siehe Seite 226). Die restlichen Interfaces, `EntityResolver` und `DTDHandler`, sind für einfache Anwendungen weniger wichtig.

```
void  endElement(String uri, String localName, String qName)
void  characters(char[] ch, int start, int length)
void  ignorableWhitespace(char[] ch, int start, int length)
void  setDocumentLocator(Locator locator)
void  startPrefixMapping(String prefix, String uri)
void  endPrefixMapping(String prefix)
void  processingInstruction(String target, String data)
void  skippedEntity(String name)
```

Beispiel für aufgerufene Methoden

Abhängig davon, was der SAX-Parser in der Eingabedatei findet, ruft er die eine oder andere dieser Methoden auf. Die XML-Datei `time-attribute.xml` (Listing 3.3) führt beispielsweise zu den folgenden Aufrufen:

```
startDocument()
startElement(, , time, {zone="GMT+1", summer="true"})
characters("
   ", 34, 4)
startElement(, , hour,})
characters("12", 44, 2)
endElement(, , hour)
characters("
   ", 53, 4)
startElement(, , minute,})
characters("50", 65, 2)
endElement(, , minute)
characters("
", 76, 1)
endElement(, , time)
endDocument()
```

Eine Anwendung implementiert das Interface `ContentHandler` in einer neuen Klasse und definiert dort alle erforderlichen Methoden. In diesen Methoden steckt die Anwendungslogik.

`DefaultHandler` stellt leere Implementierungen bereit

In der Regel interessiert sich nicht jede Anwendung für jede der oben aufgelisteten Methoden, müsste sie aber dennoch aus syntaktischen Gründen alle definieren. Das würde den Code der Handlerklasse unnötig aufblähen. Dem hilft die Klasse `DefaultHandler` ab, die selbst das Interface `ContentHandler` implementiert und leere Definitionen aller Methoden bereitstellt. Für sich gesehen ist ein `DefaultHandler` nicht sehr nützlich, vielleicht abgesehen von einfachen Testprogrammen wie dem oben gezeigten `ParseSAX`.

Konkrete Handler redefinieren einzelne Methoden nach Bedarf

Anwendungen leiten eigene Handlerklassen von `DefaultHandler` ab und redefinieren dabei nur die Methoden, für die sie sich tatsächlich interessieren. Das folgende Programm redefiniert nur `startElement` und `endElement`. Die beiden Methoden geben eingerückte Elementnamen aus.[61] Die anderen ererbten, leeren Methoden haben keine Wirkung.

[61] Der Kürze wegen ist `main` hier direkt in der Handlerklasse definiert.

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class TagsSAX extends DefaultHandler {
    private String indent = "";

    public void startElement(String uri, String localName, String qName, Attributes attri
        System.out.printf("%s<%s>%n", indent, qName);
        indent += '\t';
    }

    public void endElement(String uri, String localName, String qName) {
        indent = indent.substring(1);
        System.out.printf("%s</%s>%n", indent, qName);
    }

    public static void main(String... args) throws ParserConfigurationException, SAXExcep
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser parser = factory.newSAXParser();
        parser.parse(args[0], new TagsSAX());
    }
}
```

Listing 3.25: Abgeleiteter Handler mit Methoden für Elemente für einen SAX-Parser.

Ein Aufruf mit der Beispieldatei `time-attribute.xml` (Listing 3.3) zeigt die eingerückte Tag-Liste:

```
$ java TagsSAX time-attribute.xml
<time>
        <hour>
        </hour>
        <minute>
        </minute>
</time>
```

3.3.3 Validierung

Factory für validierenden Parser

In der Voreinstellung produziert eine `SAXParserFactory` einen minimalen SAX-Parser. Durch Konfiguration der Factory lassen sich die Parser mit zusätzlichen Fähigkeiten ausstatten.

DTD

Eine interessante Fähigkeit ist die Validierung. Der Setter `setValidating` aktiviert die Validierung gegenüber DTDs, genauso wie bei DOM-Parsern (siehe Seite 198):

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class ParseSAXValidateDTD extends DefaultHandler {

    public static void main(String... args) throws ParserConfigurationException, SAXExcep
        SAXParserFactory factory = SAXParserFactory.newInstance();
        factory.setValidating(true);
        SAXParser saxParser = factory.newSAXParser();
        saxParser.parse(args[0], new ParseSAXValidateDTD());
    }
}
```

Listing 3.26: Validieren eines XML-Dokumentes mit SAX gegenüber einer DTD.

Aufruf einer Handlermethode bei Validierungsfehler

Dieser Setter alleine hat allerdings zunächst keine erkennbare Wirkung. Das Programm akzeptiert auch ein nicht valides Dokument klaglos. Im folgenden Beispiel ist ein überzähliges Element `foobar` eingeschoben, das gemäß DTD nicht erlaubt ist:

```
$ cat time-internal-dtd-invalid.xml
<?xml version="1.0"?>
<!DOCTYPE time [
    <!ELEMENT time (hour, minute, second?)>
    <!ELEMENT hour (#PCDATA)>
    <!ELEMENT minute (#PCDATA)>
    <!ELEMENT second (#PCDATA)>
    <!ATTLIST time
        zone    CDATA           #IMPLIED
        summer  (true)          "true"
    >
]>
<time zone="GMT+1" summer="true">
    <foobar>Junk</foobar>
    <hour>12</hour>
    <minute>50</minute>
</time>
$ java ParseSAXValidateDTD time-internal-dtd-invalid.xml
$
```

Tatsächlich erkennt der Parser das unzulässige Element sehr wohl. Er meldet den Fehler durch Aufruf der Handlermethode `error`, die im Interface `ErrorHandler` definiert ist. `ParseSAXValidateDTD` erbt eine Implementierung dieser Handlermethode von der Basisklasse `DefaultHandler`, allerdings ist diese Implementierung leer! Der Fehler wird also gemeldet, die Meldung aber verworfen.

Redefinition der Handlermethode macht Fehler sichtbar

An dieser Stelle reicht es aus, `error` mit einer erkennbaren Reaktion zu redefinieren. Weiter unten wird das Interface `ErrorHandler` genauer diskutiert.

```
public void error(SAXParseException e) {
    System.out.println("error(" + e + ")");
}
```

Listing 3.27: Reaktion auf Fehler beim XML-Parsen mit SAX.

Jetzt wird der Verstoß gegen die DTD sichtbar:

```
$ java ParseSAXValidateDTD time-internal-dtd-invalid.xml
error(org.xml.sax.SAXParseException;
    systemId: time-internal-dtd-invalid.xml; lineNumber: 13; columnNumber: 10;
    Element type "foobar" must be declared.)
error(org.xml.sax.SAXParseException;
    systemId: time-internal-dtd-invalid.xml; lineNumber: 16; columnNumber: 8;
    The content of element type "time" must match "(hour,minute,second?)".)
```

XML-Schema

Parser-Factory zur Validierung mit XML-Schema

Zur Validierung gegen einem XML-Schema sind zusätzlich zu `setValidating` zwei weitere Methodenaufrufe nötig, wie das folgende Programm zeigt. `setNamespaceAware` aktiviert die zwangsläufig mit XML-Schema erforderliche Namespace-Unterstützung. Der Aufruf von `setProperty` richtet sich hier direkt an den fertigen Parser und nicht an die Factory, wie bei DOM-Parsern. Die Argumente sind genau die gleichen:[62]

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class ParseSAXValidateXSD extends DefaultHandler {
    public void error(SAXParseException e) {
        System.out.println("error(" + e + ")");
    }

    public static void main(String... args) throws ParserConfigurationException, SAXExcep
        SAXParserFactory factory = SAXParserFactory.newInstance();
        factory.setValidating(true);
        factory.setNamespaceAware(true);
        SAXParser parser = factory.newSAXParser();
        parser.setProperty("http://java.sun.com/xml/jaxp/properties/schemaLanguage",
                           "http://www.w3.org/2001/XMLSchema");
        parser.parse(args[0], new ParseSAXValidateXSD());
    }
```

[62] Das Vorgehen ist hier bei DOM und SAX nicht ganz einheitlich. Man kann auch eine SAX-Factory entsprechend konfigurieren, allerdings funktioniert das nur beim Xerces-Parser der Apache Foundation zuverlässig und scheitert möglicherweise bei Parsern anderen Fabrikats.

```
}
```

Listing 3.28: Validieren eines XML-Dokumentes mit SAX gegenüber einer XSD.

Das folgende Dokument referenziert das XML-Schema `time.xsd` (Listing 3.12), enthält aber das unzulässige Element `foobar`:

```
<?xml version="1.0"?>
<time
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="time.xsd"
    zone="GMT+1"
    summer="true">
    <foobar>Junk</foobar>
    <hour>12</hour>
    <minute>50</minute>
</time>
```

Listing 3.29: `time-xsd-invalid.xml`, XML-Dokument mit einem unzulässigen Element.

Die Validierung deckt diesen Fehler auf:

```
$ java ParseSAXValidateXSD time-xsd-invalid.xml
error(org.xml.sax.SAXParseException;
    systemId: time-xsd-invalid.xml; lineNumber: 7; columnNumber: 13;
    cvc-complex-type.2.4.a: Invalid content was found starting with element 'foobar'.
    One of '{hour}' is expected.)
$
```

Validierung gegenüber einem externen Schema

Mit einem weiteren Methodenaufruf kann man ein eigenes Schema zur Validierung heranziehen. Es spielt dann keine Rolle mehr, ob im XML-Dokument überhaupt ein Schema genannt ist. Dazu teilt man dem Parser das eigene Schema über einen zweiten `setProperty`-Aufruf mit:

```
parser.setProperty("http://apache.org/xml/properties/schema/external-noNamespaceSchemaLoc
```

`ParseSAXValidateXSD` (Listing 3.28) erwartet mit dieser Ergänzung zwei Kommandozeilenargumente, eine XML-Datei und ein XML-Schema. Das Ergebnis stimmt mit dem vorhergehenden Beispiel überein, wenn auf der Kommandozeile wieder das gleiche XML-Schema `time.xsd` (Listing 3.12) angegeben wird:

```
$ java ParseSAXValidateXSD time-xsd-invalid.xml time.xsd
error(org.xml.sax.SAXParseException;
    systemId: time-xsd-invalid.xml; lineNumber: 7; columnNumber: 13;
    cvc-complex-type.2.4.a: Invalid content was found starting with element 'foobar'.
    One of '{hour}' is expected.)
```

3.3.4 Umgang mit Text

Aufgespaltene Textknoten

Ein SAX-Parser ruft mit Textknoten den Handler

```
void  characters(char[] ch, int start, int length)
```

auf. Wie im Beispiel auf Seite 217 zu sehen ist, steckt der Textinhalt in einem Ausschnitt eines `char`-Arrays. Der Rest des Arrays ist für den Handler tabu. CDATA-Abschnitte meldet der SAX-Parser ebenfalls über `characters`-Aufrufe, sodass dafür keine Sonderbehandlung nötig ist.

`characters`-Handleraufrufe

Trotzdem ist der Umgang mit Text etwas aufwendiger, als man zunächst vielleicht annehmen würde: Der Parser ist nicht verpflichtet, Textpassagen in *einen einzigen* Aufruf des Handlers `characters` zu packen. Ein vollkommen SAX-konformer Parser könnte den Handler sogar mit jedem einzelnen Zeichen aufrufen! So extrem verhält sich der Standard-Parser in der Laufzeitbibliothek nicht, wie das Parsen des folgenden Dokuments zeigt:

```
<?xml version="1.0"?>
<text>
   <content>Das Zeichen &lt; muss in <![CDATA[XML-Dokumenten]]> entwertet werden.</conten
</text>
```

Listing 3.30: Element mit Ersatzdarstellung und CDATA-Abschnitt im Textinhalt.

Der Inhalt des Elements `content` löst dennoch fünf einzelne, direkt aufeinanderfolgende `characters`-Aufrufe aus:[63]

```
$ java PrinterSAX text.xml
startDocument()
startElement(, , text,})
```

[63] Das Programm `PrinterSAX` installiert eine Implementierung des `DefaultHandler`, der lediglich alle Methodenaufrufe mit ihren jeweiligen Argumenten protokolliert.

```
characters("
   ", 7, 4)
startElement(, , content,})
characters("Das Zeichen ", 20, 12)
characters("<", 0, 1)
characters(" muss in ", 36, 9)
characters("XML-Dokumenten", 0, 14)
characters(" entwertet werden.", 71, 18)
endElement(, , content)
characters("
", 99, 1)
endElement(, , text)
endDocument()
```

Man erkennt, dass Ersatzdarstellungen (`<`) und CDATA-Abschnitte zu einzelnen Handleraufrufen führen. Das ist etwas lästig.

Puffern aufeinanderfolgender Textfragmente

Das Problem lässt sich mit einem modifizierten `characters`-Handler beheben. Statt das Argument bei jedem Aufruf sofort zu verarbeiten, fügt der Handler neue Strings an einen internen `StringBuilder` an, der als Puffer dient:

```
private final StringBuilder buffer = new StringBuilder();

public void characters(char[] ch, int start, int length) {
    buffer.append(ch, start, length);
}
```

Listing 3.31: Puffern von Text in einem SAX-Parser.

Die eigentliche Verarbeitung läuft in einer getrennten Methode `flushCharacters` ab. Diese Methode löscht am Ende den `StringBuilder`, der damit bereit für die nächsten `characters`-Aufrufe ist.

```
private void flushCharacters() {
    if(buffer.length() == 0)
        return;
    System.out.printf("%s\"%s\"%n", indent, buffer);
    buffer.setLength(0);
}
```

Listing 3.32: Verarbeitung von gepuffertem Text in einem SAX-Parser.

Verarbeitung gepufferten Textes

Bleibt die Frage, wer für den Aufruf von `flushCharacters` verantwortlich ist. Der SAX-Parser kündigt das Ende einer Serie von `characters`-Aufrufen leider nicht an. Deshalb sollten *alle anderen* Handler als `characters` zu Beginn als Erstes `flushCharacters`

aufrufen, um eventuell gepufferte Textdaten zu verarbeiten.[64] Das folgende Programm setzt diese Idee um:

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class TextSAX extends DefaultHandler {
    private String indent = "";

    private final StringBuilder buffer = new StringBuilder();

    public void characters(char[] ch, int start, int length) {
        buffer.append(ch, start, length);
    }

    private void flushCharacters() {
        if(buffer.length() == 0)
            return;
        System.out.printf("%s\"%s\"%n", indent, buffer);
        buffer.setLength(0);
    }

    public void startElement(String uri, String localName, String qName, Attributes attri
        flushCharacters();
        System.out.printf("%s<%s>%n", indent, qName);
        indent += '\t';
    }

    public void endElement(String uri, String localName, String qName) {
        flushCharacters();
        indent = indent.substring(1);
        System.out.printf("%s</%s>%n", indent, qName);
    }

    public static void main(String... args) throws ParserConfigurationException, SAXExcep
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser parser = factory.newSAXParser();
        parser.parse(args[0], new TextSAX());
    }
}
```

Listing 3.33: Puffern von Text in einem SAX-Parser.

Whitespace

Layout als Text

Einen Blick ist der Umgang mit Layout wert. Das vorhergehende Beispiel zeigt,

[64] Geneau genommen müssen nicht *alle* Handler instrumentiert werden. Zum Beispiel kann in einem wohlgeformten XML-Dokument ein Aufruf von `endDocument` nicht direkt nach einem `characters`-Aufruf folgen.

dass ein SAX-Parser auch Text, der nur aus Zwischenraum besteht, an den `characters`-Handler weitergibt. Im Katalog der Handlermethoden (Seite 216) findet sich zwar eine Methode `ignorableWhitespace`, die aber anscheinend nicht aufgerufen wird.

Erkennen von Layout mit passender Handlermethode

Das Bild ändert sich, sobald eine DTD oder ein XML-Schema im Spiel ist. Das folgende XML-Dokument ergänzt `text.xml` (Listing 3.30) um eine interne DTD:

```
<?xml version="1.0"?>
<!DOCTYPE text [
    <!ELEMENT text (content)>
    <!ELEMENT content (#PCDATA)>
]>
<text>
   <content>Das Zeichen &lt; muss in <![CDATA[XML-Dokumenten]]> entwertet werden.</conten
</text>
```

Der Parser ruft jetzt wie erwartet `ignorableWhitespace` auf:

```
$ java PrinterSAX text-internal-dtd.xml
startDocument()
startElement(, , text,})
ignorableWhitespace("
   ", 90, 4)
startElement(, , content,})
characters("Das Zeichen ", 103, 12)
characters("<", 0, 1)
characters(" muss in ", 119, 9)
characters("XML-Dokumenten", 0, 14)
characters(" entwertet werden.", 154, 18)
endElement(, , content)
ignorableWhitespace("
", 182, 1)
endElement(, , text)
endDocument()
```

DTD oder XSD bestimmen Einstufung von Zwischenraum

Dabei spielt es keine Rolle, ob der Parser validiert oder nicht! Der Parser braucht die Dokumentdefinition hier nur, um signifikaten Zwischenraum von Layout zu unterscheiden. Das folgende Beispiel enthält beide Sorten von Whitespace:

```
<?xml version="1.0"?>
<!DOCTYPE text [
    <!ELEMENT text (content)>
    <!ELEMENT content (#PCDATA)>
]>
<text>
   <content>
   </content>
</text>
```

Listing 3.34: DTD zur Unterscheidung von Layout und signifikantem Zwischenraum.

Die DTD legt fest, dass Whitespace innerhalb von `content` als `#PCDATA` zu klassifizieren und damit signifikant ist:

```
<!ELEMENT content (#PCDATA)>
```

Sie legt auch fest, dass unter `text`-Elementen nur ein `content`-Element steht und erwähnt keinen Zwischenraum. Daher stuft der Parser Zwischenraum im `text`-Element als nicht signifikant ein und meldet ihn mit Aufrufen von `ignorableWhitespace` als Layout.

```
<!ELEMENT text (content)>
```

Die Programmausgabe macht das sichtbar:

```
$ java PrinterSAX text-blank-internal-dtd.xml
startDocument()
startElement(, , text,})
ignorableWhitespace("
   ", 90, 4)
startElement(, , content,})
characters("
   ", 103, 4)
endElement(, , content)
ignorableWhitespace("
", 117, 1)
endElement(, , text)
endDocument()
```

3.3.5 Fehlerbehandlung

Übergabe von Exceptions an Handler

Das Interface `ErrorHandler` definiert Methoden, die der SAX-Parser bei Problemen aufruft. Es gibt drei Methoden für unterschiedliche Problemkategorien:

`void fatalError(SAXParseException exception) throws SAXException`
: Dokument ist nicht wohlgeformt.

`void error(SAXParseException exception) throws SAXException`
: Dokument ist nicht valide.

`void warning(SAXParseException exception) throws SAXException`
: Nicht relevant, tritt praktisch nicht auf.

Alle drei Methoden erhalten ein Argument vom Typ `SAXParseException`, einer Checked-Exception. Dieses Vorgehen ist ungewöhnlich: Der Parser wirft die `SAXParseException`

nicht selbst, sondern übergibt sie als Argument an einen der Handler.[65] Diesem steht es frei, die Exception tatsächlich zu werfen oder anders damit umzugehen.

Handlermethoden für unterschiedliche Problemklassen

Eine redefinierte Version von `fatalError` sollte die Exception unbedingt werfen. Der Aufruf dieses Handlers bedeutet ja, dass die vorliegende Eingabe überhaupt kein XML-Dokument ist! Eine Weiterverarbeitung ist damit ziemlich aussichtslos. Besser der Parser stoppt sofort, als noch Flurschäden zu riskieren.

Nicht ganz so verfahren ist die Situation beim Aufruf von `error`. Immerhin handelt es sich um ein XML-Dokument mit bekannten Strukturmerkmalen, wenn auch nicht in der erwarteten Ausprägung. Der Parser kann seine Arbeit kontrolliert fortsetzen und wird die Handler auch weiterhin korrekt aufrufen.

Alle drei Handler können eine `SAXException` werfen, eine Basisklasse von `SAXParseException`. Zur leichteren Fehlersuche kapselt eine `SAXParseException` die Textposition des Fehlers im XML-Dokument. Sie kann mit den Gettern `getLineNumber` und `getColumnNumber` abgefragt werden.

Sinnvolle Vordefinition im `DefaultHandler`

Die Hilfsklasse `DefaultHandler` implementiert das Interface `ErrorHandler` ebenso wie das Interface `ContentHandler`. Die Methoden `error` und `warning` im `DefaultHandler` sind leer, `fatalError` wirft die übergebene `SAXParseException`.

Zusammenfassung

- XML-Dokumente sind eine system- und programmiersprachenneutrale **Textdarstellung einer Baumstruktur**.
- XML-Dokumente bestehen im Wesentlichen aus **Elementen, Attributen und Textfragmenten**.
- Eine **XML-Grammatik** regelt die zulässigen Bestandteile eines XML-Dokuments und deren Kombinationsmöglichkeiten.
- Populäre Schreibweisen von XML-Grammatiken sind **DTDs** und **XML-Schemata**.
- **Validierung** gleicht ein XML-Dokument mit einer XML-Grammatik ab.
- Ein **DOM-Parser** liest ein XML-Dokument und baut daraus ein DOM (*Document Object Model*), einen Baum von `Node`-Objekten, auf.
- `Node` und abgeleitete Klassen definieren Methoden zur **Analyse** und zur **Modifikation eines DOM**.
- Weitere Methoden erzeugen ein **neues DOM** und geben ein DOM wieder als **XML-Dokument aus**.

[65] Andere Exceptions, wie zum Beispiel eine `IOException`, wirft der SAX-Parser dagegen sofort und ohne einen Handler einzuschalten.

- **SAX-Parser** bauen kein DOM auf, sondern verarbeiten ein XML-Dokument mit **Handlern** schon **während des Einlesens**.
- SAX-Parser arbeiten im Rahmen ihrer Möglichkeiten **schneller und effizienter** als DOM-Parser.

Aufgaben

Aufgabe 1: Sortieren eines DOM

Im Allgemeinen ist die Reihenfolge der Knoten in einem XML-Dokument signifikant. Bei manchen Anwendungen spielt sie aber keine Rolle. Schreiben Sie eine Anwendung `SortXML`, die ein XML-Dokument liest, die Kindknoten auf jeder Ebene sortiert und das modifizierte Dokument wieder ausgibt. Whitespace wird generell ignoriert.

Die Reihenfolge von Knoten ist folgendermaßen geregelt:

- Knoten werden nach fallenden Typecodes sortiert (siehe Methode `getNodeType`). Das bedeutet beispielsweise, dass Textknoten vorne und Elemente hinten einsortiert werden.
- Knoten mit gleichen Typen werden nach Namen sortiert (Methode `getNodeName`).
- Knoten mit gleichen Namen werden nach Werten sortiert (Methode `getNodeValue`). Knoten ohne Wert (`null`) werden nach hinten sortiert.
- Knoten ohne Wert werden nach dem ersten unterschiedlichen Kindknoten sortiert.
- Ansonsten gelten die Knoten als gleich und können beliebig angeordnet werden.

Zur Lösung dieser Aufgabe ist die `Document`-Methode `importNode` nützlich, die einen Knoten eines Dokuments in ein anderes kopiert. Beispielsweise wird das Eingabedokument

```
<?xml version="1.0"?>
<root>
    <mm/>
    <m/>
    <mm><m/></mm>
    <sub>a c</sub>
    <sub>a b c</sub>
    <sub>a b</sub>
    z
</root>
```

sortiert zu:

```
<?xml version="1.0"?>
<root>
    z
    <m/>
    <mm/>
    <mm><m/></mm>
    <sub>a b</sub>
    <sub>a b c</sub>
    <sub>a c</sub>
</root>
```

Aufgabe 2: XML-Abbild eines Directorybaums

Schreiben Sie ein Programm, das einen Directorybaum durchläuft und ihn als XML-Dokument ausgibt. Die Elemente des XML-Dokuments entsprechen den Elementen des Directorybaums. Jedes XML-Element enthält den Namen des Filesystem-Elements als Text. Das Dokument besteht aus den folgenden XML-Elementen:

- `file` repräsentiert eine reguläre Datei.
- `directory` repräsentiert ein Directory. Es hat eine offene Anzahl Kindelemente, die dem Inhalt des Directory entsprechen.
- `unknown` repräsentiert einen anderen Filesystem-Bewohner.

Weiter gibt es die folgenden Attribute:

- `timestamp` enthält für jedes Element den Zeitpunkt der letzten Änderung als Anzahl Millisekunden (Methode `getLastModifiedTime`, siehe beispielsweise `PathInfo`).
- `length` enthält bei Dateien die Länge in Bytes.
- `exception` enthält nur bei Directories, deren Inhalt nicht lesbar ist, die Fehlerursache.[66]

Das folgende Beispiel zeigt eine mögliche Ausgabe:[67]

[66] Das Programm greift nur bei Directories auf den Inhalt zu, aber nicht bei Files und anderen Filesystem-Elementen. Daher kommt dieses Attribut nur für Directories infrage.

[67] Die Datei wurde nachträglich zur besseren Lesbarkeit hübsch formatiert. Das muss Ihre Lösung nicht tun. Am Ende dieser Aufgabe wird ein anderer Weg gezeigt, wie sich XML-Dokumente übersichtlich darstellen lassen.

```
<?xml version="1.0" encoding="UTF-8"?>
<directory timestamp="1302354462000">plugin
    <directory timestamp="1302354462000">desktop
        <file length="624" timestamp="1321565999000">sun_java.desktop</file>
        <file length="4351" timestamp="1321565999000">sun_java.png</file>
    </directory>
    <directory timestamp="1306812867000">i386
        <directory timestamp="1302354462000">ns7
            <file length="145304" timestamp="1321565999000">libjavaplugin_oji.so</file>
        </directory>
    </directory>
    <directory exception="AccessDeniedException" timestamp="1321565999000">topsecret</dir
</directory>
```

Die Struktur des XML-Dokuments regelt das folgende Schema `dirtree.xsd`. Benutzen Sie es, um die Korrektheit Ihrer Lösung zu überprüfen.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="directory">
    <xs:complexType mixed="true">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="directory"/>
        <xs:element ref="unknown"/>
        <xs:element ref="file"/>
      </xs:choice>
      <xs:attribute name="exception"/>
      <xs:attribute name="timestamp" use="required" type="xs:unsignedLong"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="unknown">
    <xs:complexType mixed="true">
      <xs:attribute name="timestamp" use="required" type="xs:unsignedLong"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="file">
    <xs:complexType mixed="true">
      <xs:attribute name="timestamp" use="required" type="xs:unsignedLong"/>
      <xs:attribute name="length" use="required" type="xs:unsignedLong"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 3.35: XML-Schema für die Struktur eines Directory-Baumes als XML-Dokument.

Grundsätzlich eignet sich jeder Texteditor zur Betrachtung des Ausgabedokuments, allerdings ist die Struktur nicht unbedingt gut erkennbar. Die meisten modernen Browser können XML-Dokumente in leidlich lesbarer Form zeigen, wenn man Vorgaben zur Darstellung beisteuert. Die folgende **CSS-Datei** `dirtree.css` (CSS = *Cascading Style Sheet*) eignet sich für die Ausgabedokumente Ihres Programms:

```
* {
    position: relative;
    display: block;
    left: 2em;
    font-weight: normal;
}

directory {
    color: blue;
    font-weight: bold;
}

file {
    color: teal;
}

unknown {
    color: lime;
}

*[exception] {
    color: red;
}
```

Listing 3.36: CSS-Datei zur Darstellung eines XML-Dokumentes mit einem Directory-Baum im Browser.

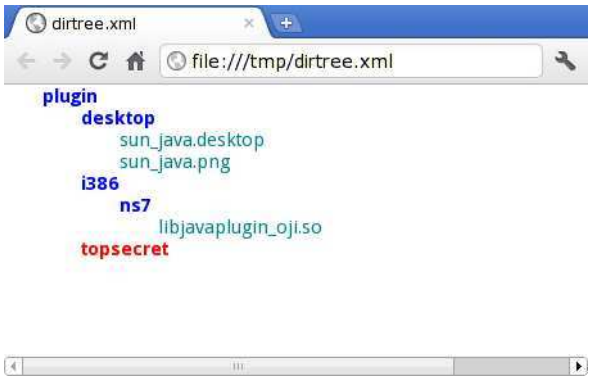
Die Verknüpfung zwischen dem XML-Dokument und der CSS-Datei stellt eine sogenannte **Processing-Instruction** her, die als zweite Zeile des XML-Dokuments, also direkt nach der obligatorischen Deklaration und vor dem Wurzelelement eingeschoben wird:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="dirtree.css"?>
<directory ...
```

Eine Processing-Instruction kann auch programmatisch mit den folgenden Anweisungen erzeugt werden:

```
ProcessingInstruction instruction =
    document.createProcessingInstruction("xml-stylesheet",
        "type=\"text/css\" href=\"dirtree.css\"");
document.appendChild(instruction);
```

Das Ergebnis zeigt sich beispielsweise im Browser „Chrome“ so:



Aufgabe 3: SAX-Parser

Ein wesentlicher Vorteil von SAX-Parsern gegenüber DOM-Parsern ist die Möglichkeit, das Einlesen schon vor dem Ende des Dokuments abzubrechen. Die Mühe lohnt sich nicht bei kleinen Dokumenten, wirkt sich aber bei langen und möglicherweise endlosen Dokumenten deutlich aus.[68]

Daten für Übungszwecke können Sie mit dem folgenden Programm erzeugen:

```
import static java.lang.System.*;

public class XMLFactors {
    public static void main(String... args) {
        int upto = Integer.parseInt(args[0]);
        out.printf("<?xml version=\"1.0\"?>%n");
        out.printf("<root>%n");
        for(int n = 1; n < upto; n++) {
            out.printf("\t<number>%n");
            out.printf("\t\t<value>%d</value>%n", n);
            int rest = n;
            int f = 2;
            while(rest > 1)
                if(rest%f == 0) {
                    out.printf("\t\t<factor>%d</factor>%n", f);
                    rest /= f;
                }
                else
                    f++;
            out.printf("\t</number>%n");
        }
        out.printf("</root>%n");
```

[68] Ein „endloses“ Dokument stammt nicht aus einer Datei, sondern wird beispielsweise von einem Sensor generiert, der fortlaufend XML-formatierte Messwerte über eine Netzwerkverbindung liefert.

```
    }
}
```

Listing 3.37: Generiert ein einfaches, aber langes XML-Dokument.

Dieses Programm erwartet auf der Kommandozeile eine positive Zahl n und produziert dann ein XML-Dokument, das für alle natürlichen Zahlen von 1 bis n die Primfaktoren angibt:[69]

```
java XMLFactors 1000000 > long.xml
```

Das Resultat ist ein langes XML-Dokument mit einem recht simplen Aufbau, der am Anfang der Datei auch ohne Grammatik leicht zu erkennen ist:

```
<?xml version="1.0"?>
<root>
    <number>
        <value>1</value>
    </number>
    <number>
        <value>2</value>
        <factor>2</factor>
    </number>
    <number>
        <value>3</value>
        <factor>3</factor>
    </number>
    <number>
        <value>4</value>
        <factor>2</factor>
        <factor>2</factor>
    </number>
    ...
```

Stellen Sie sich nun vor, die `factor`-Elemente in dieser Datei wären „Messwerte“, unter denen ein bestimmtes Muster gesucht wird. Schreiben Sie ein Programm `SAXFactorScanner`, das in dieser Datei nach dem n-ten Vorkommen eines bestimmten Primfaktors sucht und die zugehörige Zahl ausgibt. Das 10000. Vorkommen des Primfaktors 61 findet man beispielsweise in der Zahl 600057. Diese Antwort liefert das Programm:

```
$ java SAXFactorScanner 10000 61 < long.xml
value=600057
```

[69] Eine komprimierte Fassung dieses Dokuments steht auf `http://sol.cs.hm.edu/4129/long.xml.7z` zum Download zur Verfügung.

Dazu müssen viele Megabyte XML-Daten geparst werden. Schnell sollte dagegen der folgende Aufruf ablaufen:

```
$ java SAXFactorScanner 3 2 < long.xml
value=4
```

Ein DOM-Parser würde die Datei `long.xml` komplett in den Speicher holen. Das dauert einerseits eine ganze Weile. Darüber hinaus beanspruchen die Objekte, aus denen das DOM aufgebaut ist, einigen Platz auf dem Heap. Zu allem Überfluss wäre der ganze Aufwand beim zweiten Aufruf weitgehend unnötig, weil das Ergebnis schon nach ein paar Zeilen feststeht und der Löwenanteil der Eingabedatei keine Rolle mehr spielt.

Kapitel

4 Rekursion

Lernziele

In diesem Kapitel lernen Sie

- wie **rekursiver Code** komplett ohne Schleifen auskommt, welche Vorteile und auch welchen Preis das hat.
- nach welchen Kriterien Sie verschiedene **Arten der Rekursion** unterscheiden können und welche charakteristischen Eigenschaften die einzelnen Rekursionsarten aufweisen.
- einige typische Probleme kennen, die sich gut mit **Rekursion lösen** lassen.
- wie Sie rekursive Programme durch **Memoizing** beschleunigen können und welche Rolle dabei **Weak-References** spielen.

Rekursion ist eine Art Kontrollstruktur, vergleichbar mit `while`- und `for`-Schleifen. Fast alle Programmiersprachen unterstützen Rekursion, darunter auch Java. Auf den ersten Blick wirkt rekursiver Code manchmal fremdartig, aber mit etwas Übung lassen sich damit oft elegante und kompakte Lösungen formulieren.[1]

Syntaktisch ist Rekursion an Methoden erkennbar, die sich selbst aufrufen.[2] Wie bei allen Methodenaufrufen wird auch bei einem Selbstaufruf der komplette „Ablaufzustand“ des Aufrufers bis zur Rückkehr der aufgerufenen Methode eingefroren. Zum Ablaufzustand zählen die Stelle des Aufrufs im Code sowie die Werte aller lokalen Variablen und Parameter. Bei jedem Selbstaufruf einer rekursiven Methode sichert das Laufzeitsystem den Ablaufzustand in einer verborgenen Datenstruktur, dem Laufzeitstack der JVM, und restauriert ihn nach der Rückkehr wieder. Der

[1] Es gibt Programmiersprachen (Lisp, Scheme, Haskell und viele weitere funktionale Programmiersprachen), in denen Rekursion das vorherrschende Sprachmittel ist. Dort wirken eher Schleifen wie Fremdkörper.

[2] Der Selbstaufruf kann auch mittelbar über andere Methoden erfolgen und ist dann nicht ganz so offensichtlich. Das ändert nichts am Prinzip. Die Laufzeitsysteme praktisch aller Programmiersprachen kommen mit solchen Selbstaufrufen zurecht.

Anwender muss sich um den Laufzeitstack nicht kümmern,[3] denn er unterliegt komplett der Verwaltung der JVM. Nutzt man diese „kostenlose“ Verwaltung geschickt aus, dann kommt man zu den angesprochenen eleganten und kompakten Lösungen.

Manche Probleme bieten sich geradezu für eine rekursive Lösung an. Dazu gehören zum Beispiel Operationen mit rekursiven Datenstrukturen,[4] wie Listen oder Bäume. Auch der Ansatz *divide and conquer* lässt sich gut rekursiv umsetzen. Dabei wird ein komplexes Problem so lange in kleinere Probleme zerlegt, bis die verbleibenden Restprobleme auf eine überschaubare Größe geschrumpft sind. Die Gesamtlösung ergibt sich aus der Kombination der Teillösungen in der umgekehrten Reihenfolge der Aufteilung.

Vererbung, dynamisches Binden, Datenkapselung und andere Konzepte der objektorientierten Programmierung spielen in diesem Kapitel keine große Rolle. Im Vordergrund steht bei Rekursion die Steuerung des Programmablaufs und weniger die Modellierung von Daten, abgesehen von Baumstrukturen. Dieses Kapitel ist nicht Java-spezifisch.

4.1 Arbeitsweise

4.1.1 Selbstaufruf und Abbruchkriterium

Selbstaufruf als syntaktisches Merkmal

Das Merkmal der Rekursion ist der Selbstaufruf einer Methode. Die folgende Methode `m` ist ein äußerst einfaches Beispiel einer rekursiven, wenn auch nutzlosen Methode:

```
void m() {
    m();
}
```

Listing 4.1: Endlosrekursive Methode.

Endlosrekursion

Ein Aufruf von `m` kehrt erst dann zurück, wenn der Methodenrumpf ausgeführt ist. Im Rumpf steht ein Aufruf von `m` selbst. Dieser kehrt zurück, sobald ein wei-

[3] Der Anwender *kann* den Laufzeitstack überhaupt nicht direkt auslesen oder gar manipulieren.

[4] Eine „rekursive“ Datenstruktur besteht aus Teilen, die den gleichen Aufbau wie die gesamte Struktur haben. Der Rest einer Liste ohne das erste Element ist zum Beispiel selbst eine, wenn auch kürzere, Liste. Auch ein Teil eines Baums, der am Wurzelknoten hängt, ist wieder ein Baum. Ein Bruch ist dagegen keine rekursive Datenstruktur, weil weder Zähler noch Nenner Brüche sind.

terer Aufruf von `m` beendet ist. Das Spiel wiederholt sich immer wieder. Der erste, „oberste“ Aufruf wird in diesem Beispiel *nie* zurückkehren, es kommt zur **Endlosrekursion**.

Absturz nach zu vielen Selbstaufrufen

Das folgende Programm ruft `m` auf. Es bricht schnell mit einem `StackOverflowError` ab.

```
public class EndlessRecursion {
    void m() {
        m();
    }

    public static void main(String... args) {
        new EndlessRecursion().m();
    }
}
```

Listing 4.2: Aufruf einer endlosrekursiven Methode.

Die Ausgabe auf der Konsole wiederholt immerfort die gleiche Zeile:

```
$ java EndlessRecursion
Exception in thread "main" java.lang.StackOverflowError
        at EndlessRecursion.m(EndlessRecursion.java:14)
        at EndlessRecursion.m(EndlessRecursion.java:14)
        at EndlessRecursion.m(EndlessRecursion.java:14)
        ...
$
```

Protokoll der Selbstaufrufe

Die Rekursion läuft in der Praxis nicht endlos, sondern führt zu einem Laufzeitfehler, das heißt zum Absturz des Programms. Die Selbstaufrufe werden sichtbar, wenn man `m` folgendermaßen um Ausgaben erweitert:

```
void m() {
    System.out.println("m() called");
    m();
    System.out.println("m() returns");
}
```

Listing 4.3: Endlosrekursive Methode mit Protokoll der Aufrufe.

Am Absturz ändert das nichts. Die Methodenaufrufe werden protokolliert, aber die Ausgaben zur Methodenrückkehr bleiben aus:

```
$ java EndlessRecursion
m() called
m() called
...
m() called
m() called
Exception in thread "main" java.lang.StackOverflowError
        at EndlessRecursion.m(EndlessRecursion.java:18)
        at EndlessRecursion.m(EndlessRecursion.java:18)
        at EndlessRecursion.m(EndlessRecursion.java:18)
        ...
$
```

Abbruchbedingung

Abbruchkriterium stoppt Selbstaufrufe

Jede rekursive Methode muss irgendwann damit *aufhören*, sich selbst aufzurufen, um Endlosrekursion und damit den Programmabsturz zu verhindern. Unverzichtbar ist daher eine **Abbruchbedingung**, die die Selbstaufrufe stoppt.

- Solange die Abbruchbedingung nicht erfüllt ist, ruft sich die Methode aufs Neue selbst auf.
- Wenn die Abbruchbedingung erfüllt ist, kehrt die Methode sofort zurück, *ohne* sich selbst noch einmal aufzurufen.

Im folgenden Beispiel zählt die Objektvariable `remaining` mit, wie viele rekursive Selbstaufrufe von `m` (noch) gefordert sind.

- Wenn noch Aufrufe anstehen (`remaining > 0`), ruft sich `m` (rekursiv) selbst auf. Zur Buchführung verringert `m` die Variable `remaining` direkt vor dem Selbstaufruf um 1.
- Ansonsten (`remaining == 0`) kehrt `m` ohne Selbstaufruf zurück.

```
public class LimitedRecursion {
    private int remaining = 10;

    void m() {
        System.out.printf("m() called, remaining = %d%n", remaining);
        if(remaining > 0) {
            remaining--;
            m();
        }
        System.out.println("m() returns");
    }

    public static void main(String... args) {
        new LimitedRecursion().m();
```

```
    }
}
```

Listing 4.4: Rekursion mit einem Zähler für die Anzahl verbleibender Aufrufe.

Initialisiert man `remaining` beispielsweise mit 10, dann laufen zehn Selbstaufrufe von `m` ab, von denen jeder wieder zurückkehrt. Zusammen mit dem ersten, direkten Aufruf von `m` in `main` beobachtet man insgesamt elf Aufrufe. Das Programm endet regulär:[5]

Abgezählte Selbstaufrufe und Rückkehr

```
$ java LimitedRecursion
m() called, remaining = 10
    m() called, remaining = 9
        ...
            m() called, remaining = 1
                m() called, remaining = 0
                m() returns
            m() returns
        ...
    m() returns
m() returns
$
```

Merkmale der Rekursion

Die beiden entscheidenden Komponenten einer rekursiven Methode sind also

1. ein Selbstaufruf und
2. eine Abbruchbedingung.

Diese Vorgaben alleine garantieren noch nicht, dass die Abbruchbedingung tatsächlich jemals erfüllt wird.[6] Jeder Methodenaufruf muss also auch in irgendeiner Form einen Beitrag zum Erreichen der Abbruchbedingung leisten.[7]

Bewegung zum Abbruchkriterium hin

[5] Die Einrückung in der Ausgabe erfordert etwas zusätzlichen Code, der hier nicht abgedruckt ist.

[6] Ohne die Vorgaben *kann* Rekursion allerdings nicht funktionieren.

[7] In einfachen Fällen, wie im vorhergehenden Beispiel, ist durch Anschauung unmittelbar einsichtig, dass die Methode unweigerlich auf das Abbruchkriterium zusteuert und es folglich auch zwangsläufig über kurz oder lang erreicht. Bei komplizierteren Programmen ist „Anschauung“ kein zuverlässiger Ratgeber. Perfekt wäre ein formaler Nachweis, dass der Code die Abbruchbedingung irgendwann erfüllen *muss*. In der Praxis lässt sich dieser Nachweis nur selten mit akzeptablem Aufwand führen.

4.1.2 Parameter und Ergebnis

Inkarnationen von Parametern

Parameter und lokale Variablen auf jeder Aufrufebene

Die Parameter einer Methode werden beim Aufruf angelegt und mit Argumentwerten initialisiert. Sie „leben“ für die Dauer des Aufrufs, das heißt bis zur Rückkehr dieses Aufrufs. Das gilt für alle Methoden, ob rekursiv oder nicht. Das entscheidende Merkmal einer rekursiven Methode ist, dass es zu einem einzigen Zeitpunkt *mehrere* Aufrufe gibt, die alle noch nicht zurückgekehrt sind. Dabei verfügt jeder einzelne rekursive Aufruf über seinen eigenen Satz Parameter, unabhängig von den anderen rekursiven Aufrufen. Obwohl die Parameter einer rekursiven Methode nur einmal im Quelltext genannt sind, können zur Laufzeit zum gleichen Zeitpunkt viele unabhängige Inkarnationen existieren.

Steuerung der Rekursion mit Parameter

Die Methode `m` im folgenden Beispiel zählt die Anzahl der verbleibenden Selbstaufrufe nicht in einer Objektvariablen, sondern im *Parameter* `remaining` mit. Die Methode überprüft `remaining` und übergibt gegebenenfalls den um 1 verringerten Wert an den nächsten Aufruf:

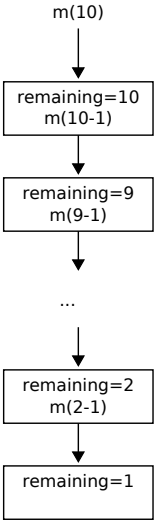
```
public class ParameterRecursion {
    void m(final int remaining) {
        System.out.printf("m(%d) called%n", remaining);
        if(remaining > 0)
            m(remaining - 1);
        System.out.printf("m(%d) returns%n", remaining);
    }
}
```

Listing 4.5: Rekursion mit der Anzahl verbleibender Aufrufe als Parameter.

Die Ausgabe zeigt die fallenden Parameterwerte beim Aufruf von `m(10)`.

```
m(10) called
    m(9) called
        ...
            m(1) called
                m(0) called
                m(0) returns
            m(1) returns
        ...
    m(9) returns
m(10) returns
```

Die folgende Skizze zeigt einen Schnappschuss der geschachtelten Aufrufe zu dem Zeitpunkt, an dem die Abbruchbedingung erfüllt ist. Jedes Kästchen stellt einen Aufruf von `m` mit seinem eigenen Parameter `remaining` dar.



In diesem Beispiel kommen keine lokalen Variablen vor. Für sie gelten aber die gleichen Regeln wie für Parameter: Jeder einzelne rekursive Aufruf verfügt über einen eigenen Satz aller lokaler Variablen, die unabhängig von den Inkarnationen der gleichen lokalen Variablen in vorausgegangenen und nachfolgenden rekursiven Aufrufen sind.

Ergebnisrückgabe

Akkumulieren des Ergebnisses

Wie alle Methoden können auch rekursive Methoden ein Ergebnis zurückliefern. Die folgende Methode `m` addiert 1 zum Ergebnis des untergeordneten rekursiven Aufrufs und gibt selbst die Summe zurück. Ganz unten, am Rekursionsende, ist das Ergebnis 0, weil kein rekursiver Aufruf mehr erfolgt. Das Ergebnis eines Aufrufs entspricht also der Anzahl der rekursiven Aufrufe und hat damit den gleichen Wert wie das Argument des Aufrufs:

```
public class ResultRecursion {
    int m(final int remaining) {
        System.out.printf("m(%d) called%n", remaining);
        final int result;
        if(remaining > 0)
            result = 1 + m(remaining - 1);
        else
            result = 0;
        System.out.printf("m(%d) returns %d%n", remaining, result);
        return result;
    }
```

```
    public static void main(String... args) {
        ResultRecursion recursion = new ResultRecursion();
        System.out.println(recursion.m(10));
    }
}
```

Listing 4.6: Ergebnisrückgabe aus rekursiven Aufrufen.

Die Ausgabe des Aufrufs `m(10)` zeigt den Ablauf:

```
$ java ResultRecursion
m(10) called
    m(9) called
        ...
            m(1) called
                m(0) called
                m(0) returns 0
            m(1) returns 1
        ...
    m(9) returns 9
m(10) returns 10
10
```

Zahlensumme

Beispiel Zahlensumme

Die vorhergehende Methode ist nicht sonderlich interessant. Addiert man aber statt einer 1 die *Anzahl der verbleibenden Aufrufe* (`remaining`), dann ändert sich das Bild.

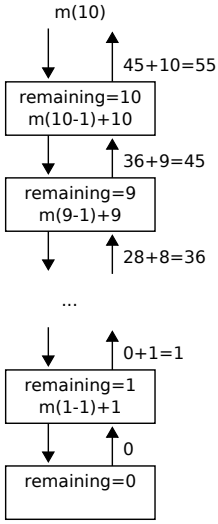
```
public class Sum {
    int m(final int remaining) {
        System.out.printf("m(%d) called%n", remaining);
        final int result;
        if(remaining > 0)
            result = remaining + m(remaining - 1);
        else
            result = 0;
        System.out.printf("m(%d) returns %d%n", remaining, result);
        return result;
    }

}
```

Listing 4.7: Rekursive Berechnung der Zahlensumme.

Der Methodenaufruf `m(`n`)` addiert jetzt bei der Rückkehr aus der Rekursion zur 0 die 1, dann 2, dann 3 und so weiter bis n. Das Ergebnis ist die Summe $s(n) = 0 + 1 + 2 + ... + n$ aller Zahlen von 1 bis n.

```
$ java Sum
m(10) called
    m(9) called
        ...
            m(2) called
                m(1) called
                    m(0) called
                    m(0) returns 0
                m(1) returns 1
            m(2) returns 3
        ...
    m(9) returns 45
m(10) returns 55
55
```

Die folgende Skizze zeigt die Argmente und Ergebnisse:



Die Methode implementiert nichts anderes als die induktive Definition der Zahlensumme:

Rekursion als mathematische Induktion

- Induktionsanfang: Für $n = 0$ ist Summe $s(0) = 0$.
- Induktionsschritt: Für $n > 0$ ist $s(n)$ um n größer als die Summe der vorhergehenden $n - 1$ Zahlen, also $s(n) = n + s(n-1)$.

Der Code spiegelt diese Definition direkt wider.

4.1.3 Indirekte Rekursion

Gegenseitiger Aufruf zweier rekursiver Methoden

Auch zwei oder mehr Methoden, die sich gegenseitig aufrufen, sind rekursiv. Man spricht in diesem Fall von **indirekter Rekursion**.

Im folgenden Beispiel stellen die beiden Methoden `even` und `odd` zusammen fest, ob eine Zahl gerade oder ungerade ist.[8] Die beiden Methoden kennen das Ergebnis für 0 und wissen außerdem, dass sich gerade und ungerade Zahlen abwechseln. Bei positiven Werten rufen sie die jeweils andere Methode mit dem dekrementierten Argument auf:

```
public class EvenOdd {
    boolean even(final int n) {
        return n == 0? true: odd(n - 1);
    }

    boolean odd(final int n) {
        return n == 0? false: even(n - 1);
    }

    public static void main(String... args) {
        EvenOdd evenOdd = new EvenOdd();
        System.out.println(evenOdd.even(Integer.parseInt(args[0])));
    }
}
```

Listing 4.8: Indirekt-rekursive Methoden.

4.1.4 Laufzeitstack

Stack als Datenspeicher für offene Methodenaufrufe

Jeder Methodenaufruf, ob rekursiv oder nicht, verwendet einen eigenen Satz Parameter und lokaler Variablen. Diese Variablen stehen zur Verfügung, bis der Aufruf endet und zurückkehrt. Aufrufe untergeordneter Methoden unterbrechen den Ablauf des Aufrufers, aber nach Rückkehr untergeordneter Aufrufe wird der Aufrufer mit denselben Variablen fortgesetzt.

Stackframes

Inhalt eines Stackframes

Das Laufzeitsystem stapelt die Daten eines Methodenaufrufs auf dem **Laufzeitstack**. Der Speicherbereich auf dem Laufzeitstack, der alle Daten eines unterbrochenen Methodenaufrufs aufnimmt, wird als **Stackframe** bezeichnet. Ein Stackframe enthält unter anderem

[8] Diese Definition ist ausgesprochen ineffizient und in der Praxis kaum brauchbar.

- die Parameter mit ihren Werten,
- die lokalen Variablen mit Werten,
- der Ort des Aufrufs und
- die noch offenen Berechnungen, falls der Methodenaufruf Teil eines Ausdrucks ist.

Nehmen Sie beispielsweise an, dass die folgende Methode `m` mit dem Argument 10 aufgerufen wird:

```
int m(final int remaining) {
    final int result;
    if(remaining > 0)
        result = remaining + m(remaining - 1);
    else
        result = 0;
    return result;
}
```

Listing 4.9: Rekursive Methode.

Direkt vor dem rekursiven Aufruf in der mittleren Codezeile der Methode enthält der Stackframe die folgenden Informationen:

- den Parameter `remaining` mit dem Wert 10,
- die lokale Variable `result`, noch ohne Wert,
- den Wert 10 als ersten Operanden der Addition zum späteren Abschluss der Addition,
- den zweiten Operanden der Addition als Punkt zur Fortsetzung nach Rückkehr des unmittelbar bevorstehenden, rekursiven Aufrufs.

Stacküberlauf

Begrenzter Laufzeitstack

Der Laufzeitstack hat eine begrenzte Größe und kann nicht beliebig viele Stackframes aufnehmen. Aus der Begrenzung ergibt sich eine maximale Anzahl geschachtelter Methodenaufrufe. Diese Anzahl schwankt, weil die Stackframes nicht alle gleich groß sind.

Das folgende Programm liefert einen Anhaltspunkt für die Größenordnung der maximalen Schachtelungstiefe von Methodenaufrufen:

```
public class MaxCalls {
    private int depth = 0;
```

```
    void m() {
        depth++;
        m();
    }

    public static void main(String... args) {
        MaxCalls calls = new MaxCalls();
        try {
            calls.m();
        }
        catch(StackOverflowError error) {
            System.out.println(calls.depth);
        }
    }
}
```

Listing 4.10: Abschätzung der maximalen Rekursionstiefe.

Die Ausgabe auf dem System des Autors (Java SE 7, 32-Bit, Linux) zeigt:

```
$ java MaxCalls
7579
```

Auskunft über voreingestellte Stackgröße

Die voreingestellte Größe des Laufzeitstacks hängt von der konkreten Java-Implementierung ab. Auskunft gibt der Aufruf mit dem Schalter `-XshowSettings:vm`:

```
$ java -XshowSettings:vm
VM settings:
    Stack Size: 320.00K
    Max. Heap Size (Estimated): 448.00M
    Ergonomics Machine Class: server
    Using VM: Java HotSpot(TM) Server VM
```

In diesem Beispiel beträgt die voreingestellte Stackgröße 320 kB. Man könnte versuchen, aus dem Ergebnis von `MaxCalls` die Größe eines Stackframes zu berechnen. Das Ergebnis ist allerdings nicht sehr aussagekräftig, weil eine JVM mehrere Threads gleichzeitig abwickelt, deren Laufzeitstacks sich diesen Speicherbereich teilen.

Anpassung der Stackgröße

Die Stackgröße kann beim Start der JVM mit dem Schalter `-Xss` festgelegt werden. Der Schalter akzeptiert als Argument eine Anzahl mit dem Suffix `k` (für Kilobyte), `M` (Megabyte) und `G` (Gigabyte).[9] Ein neuer Start von `MaxCalls` mit einem Laufzeitstack von 256 MB erlaubt eine weit höhere Rekursionstiefe:

[9] Diese Angabe kann natürlich nur im Rahmen der tatsächlich verfügbaren Ressourcen umgesetzt werden. Bei zu hohen Vorgaben startet die JVM überhaupt nicht und bricht stattdessen sofort mit dem Fehler `Invalid thread stack size` ab.

```
$ java -Xss256M MaxCalls
33470794
```

4.1.5 Rekursive Datenstrukturen

Datenstrukturen aus wiederkehrende Bausteinen

Eng verwandt mit rekursiven Methoden sind rekursive Datenstrukturen. Sie bestehen aus einer offenen Anzahl gleichartiger Elemente, die durch Referenzen verknüpft sind. Typische Beispiele sind verkettete Listen und Bäume. Rekursive Datenstrukturen lassen sich gut mit rekursiven Methoden verarbeiten.

Beispiele aus anderen Kapiteln dieses Buches sind:

- die Methode `read` in `StringIterableReader` zum Vorauslesen in einem String,
- die Methode `walk` zum Durchlaufen eines DOM in `WalkDOM`,
- Methoden in Umsetzungen des Decorator-Patterns, wie Pizzas (Seite 623) und I/O-Filterketten (Seite 51).

4.2 Rekursion und Iteration

Grundsätzliche Äquivalenz

Rekursion und Iteration, das heißt Schleifenkonstrukte, sind äquivalent. Prinzipiell lässt sich daher rekursiver Code immer in Schleifen umformen und umgekehrt.[10] Diese grundsätzliche Einsicht sagt allerdings nichts über die Lesbarkeit, den Umfang und die Effizienz des Ergebnisses aus. Lineare Rekursion (Seite 252) kann man beispielsweise oft durch eine einfachere und effizientere Iteration ausdrücken. Dagegen fällt bei verzweigter Rekursion (Seite 255) eine iterative Lösung meist voluminös – und entsprechend schwer handhabbar – aus. Geschachtelte Rekursion (Seite 259) kann praktisch überhaupt nicht mehr in eine brauchbare iterative Lösung umgeschrieben werden.

Endrekursion als einfacher Fall

An einer besonders einfachen Art der Rekursion, der „Endrekursion“, lässt sich die quasi-mechanische Transformation in eine Iteration anschaulich zeigen.

4.2.1 Endrekursion

Ergebnisparameter

Rekursiver Aufruf als letzte Operation

Eine Methode ist **endrekursiv**, wenn sie nach einem rekursiven Aufruf keinen wei-

[10] Es gibt keine Probleme, die sich *nur* mit Rekursion oder *nur* mit Schleifen lösen ließen.

teren Code mehr ausführt. Die Methode im Programm `Sum` (Listing 4.7) ist *nicht* endrekursiv, weil nach der Rückkehr aus dem rekursiven Aufruf noch `remaining` zum Ergebnis addiert wird. In der folgenden Fassung ist die Methode `sum` dagegen endrekursiv, weil das Ergebnis des rekursiven Aufrufs unverändert das Ergebnis des ganzen Aufrufs ist:

```
public class SumTail {
    int sum(final int n, final int result) {
        if(n > 0)
            return sum(n - 1, result + n);
        else
            return result;
    }

    public static void main(String... args) {
        System.out.println(new SumTail().sum(10, 0));
    }
}
```

Listing 4.11: Endrekursive Berechnung der Zahlensumme mit Akkumulator-Parameter für das Ergebnis.

Zusätzlicher Ergebnisparameter

Die Endrekursion hat ihren Preis: Die Methode muss um einen zweiten Parameter (`result`) erweitert werden, der das Ergebnis aufnimmt. Im Gegensatz zur vorhergehenden Fassung `Sum` „akkumuliert“ `SumTail` das Ergebnis schon bei den rekursiven Aufrufen und nicht erst bei der Rückkehr aus der Rekursion. Der zusätzliche Parameter sammelt dabei alle Teilergebnisse auf. Im letzten, untersten rekursiven Aufruf enthält `result` bereits das Endergebnis, das die rekursiven Aufrufe nur noch unverändert zurückgibt. Beim Aufstieg aus der Rekursion wird kein Code mehr ausgeführt, es finden keine Berechnungen mehr statt.

Alle Berechnungen beim „Abstieg“

Die eigentliche Arbeit leistet die Methode bei der Berechnung der Argumente des rekursiven Aufrufs. Das wird deutlich sichtbar, wenn man die Argumentwerte vor dem rekursiven Aufruf in temporären lokalen Variablen zwischenspeichert, wie in der folgenden Fassung:

```
public class SumTailTempVars {
    int sum(final int n, final int result) {
        if(n > 0) {
            final int tempN = n - 1;
            final int tempResult = result + n;
            return sum(tempN, tempResult);
        }
        else
            return result;
    }

    public static void main(String... args) {
```

```
        System.out.println(new SumTailTempVars().sum(10, 0));
    }
}
```

Listing 4.12: Endrekursive Methode mit Zwischenspeichern der Argumentwerte in temporären Variablen.

Untersucht man den Ablauf von `SumTail` (Listing 4.11) oder `SumTailTempVars` (Listing 4.12) genau, so wird der Methodenrumpf einmal ausgeführt, dann im nächsten rekursiven Aufruf erneut und so fort, bis die Abbruchbedingung erfüllt ist. Danach geschieht nichts mehr, außer der Rückgabe des Ergebnisses durch alle Aufrufebenen. Die Informationen in den Stackframes, die bei den rekursiven Aufrufen auf dem Stack gestapelt wurden, werden bei der Rückkehr nicht mehr gebraucht.

Stackframes überflüssig

Genau den gleichen Ablauf erreicht man mit einer Schleife, die am Ende die temporären Variablen an die Parameter zuweist und sie damit auf die Werte aktualisiert, die vorher an den nächsten rekursiven Aufruf übergeben wurden:

```
public class SumLoopTempVars {
    int sum(int n, int result) {
        while(n > 0) {
            final int tempN = n - 1;
            final int tempResult = result + n;
            n = tempN;
            result = tempResult;
        }
        return result;
    }

    public static void main(String... args) {
        System.out.println(new SumLoopTempVars().sum(10, 0));
    }
}
```

Listing 4.13: Aktualisieren von Variablen entsprechend zu den Argumentwerten der rekursiven Fassung.

Transformationsschema

Diese Schema lässt sich verallgemeinern. Ein endrekursive Methode mit Argumenten `expression...` im rekursiven Aufruf hat die folgende Gestalt:

Mechanisches Umschreiben der Endrekursion

```
type method(param1, param2, ..., result) {
    if(condition) {
        // Code ...
        return method(expression1, expression2, ..., expressionX);
    }
    else
```

```
        return result;
}
```

Sie lässt sich *immer* in die folgende iterative Methode umschreiben:

```
type method(param1, param2, ..., result) {
    while(condition) {
        // Code ...
        final temp1 = expression1;
        final temp2 = expression2;
        ...
        final tempX = expressionX;
        param1 = temp1;
        param2 = temp2;
        ...
        result = tempX;
    }
    return result;
}
```

Iterative Version ohne Stackverbrauch

Die iterative Fassung legt keine Stackframes an und „verbraucht“ keinen Laufzeitstack. Anders als die rekursive Fassung droht deshalb keine Gefahr des Stacküberlaufs. Abgesehen davon läuft die Schleife etwas schneller, weil das Anlegen von Stackframes Zeit kostet.[11]

Parallele Parameterzuweisungen

Simulation der parallelen Zuweisung der Argumente

Man könnte versucht sein, den Weg über die Zwischenvariablen `temp`... einzusparen und die den Parametern zugeordneten Variablen sofort mit den Ausdrücken `expression`... zu aktualisieren. Diese Kürzung könnte allerdings falsche Ergebnisse liefern, weil Argumente quasi *gleichzeitig* an Parameter übergeben werden. Dagegen läuft eine Folge von Wertzuweisungen immer *sequenziell* ab.

Zur Illustration dient das nächste Programm, das den größten gemeinsamen Teiler (*Greatest Common Divisor*, GCD) von zwei positiven Zahlen endrekursiv berechnet:

```
import static java.lang.System.*;

public class EuclidGCD {
    int gcd(int n, int m) {
        if(n%m > 0)
```

[11] Die JVM wickelt Methodenaufrufe zwar effizient ab, aber nicht zum Nulltarif.

```
            return gcd(m, n%m);
        return m;
    }

    public static void main(String... args) {
        EuclidGCD euclid = new EuclidGCD();
        out.println(euclid.gcd(9, 6));
    }
}
```

Listing 4.14: Rekursive Berechnung des ggT mit Euklids Algorithmus.

Die „mechanische“ Eliminierung der Endrekursion liefert die folgende iterative Fassung.

```
int gcd(int n, int m) {
    while(n%m > 0) {
        final int tempN = m;
        final int tempM = n%m;
        n = tempN;
        m = tempM;
    }
    return m;
}
```

Listing 4.15: Iterative Fassung der Berechnung des ggT mit Euklids Algorithmus.

Hier sind die temporären Variablen notwendig, weil die direkte Zuweisung an die Parameter ein falsches Ergebnis liefern würde:

```
int gcd(int n, int m) {
    while(n%m > 0) {
        n = m;
        m = n%m;  // falsch, n bereits aktualisiert
    }
    return m;
}
```

Listing 4.16: Fehlerhafte iterative Fassung von Euklids Algorithmus ohne temporäre Variablen.

Auch vertauschte Wertzuweisungen lösen das Problem der sequenziellen Aktualisierung nicht.

4.2.2 Eliminierung der Endrekursion

Die Umformung von Endrekursion in Iteration kann mechanisch ausgeführt wer-

Keine automatische Umformung durch den Java-Compiler

den. Der Zweck der Methode spielt dabei keine Rolle, entscheidend ist nur die Endrekursion selbst. Manche Compiler erkennen diese Konstellation und wenden die Umformung, die als *tail-recursion elimination* bezeichnet wird, automatisch an.

Die Spezifikation von Java regelt nicht, wie mit Endrekursion umgegangen werden soll. Die Entscheidung ist Compilerherstellern freigestellt.[12] Der Java-Compiler im Oracle JDK Version 7 eliminiert Endrekursion nicht.[13] Gründe für diese Entscheidung sind:

- Die Transformation löst den Bezug zwischen dem generierten Bytecode und dem ursprünglichen Quelltext bis zu einem gewissen Grad auf. Bei der Fehlersuche hat ein Debugger dann möglicherweise Probleme, den tatsächlichen Ablauf dem Quelltext zuzuordnen.
- Den Vorteilen der Eliminierung von Endrekursion stehen die Kosten der vorher nötigen Analyse des Quelltexts gegenüber. Rekursion ist in Java-Programmen nicht unbedingt vorherrschend, deshalb sind andere Optimierungen aussichtsreicher und damit rentabler.

4.3 Klassifizierung

Anzahl und Anordnung rekursiver Aufrufe

Endrekursion lässt sich mechanisch in Iteration umschreiben. Bei komplizierteren rekursiven Methoden erfordert die Transformation aber zunehmend mehr Aufwand. Wann sollte man sich noch um eine iterative Lösung bemühen und wann lohnt sich der Aufwand eher nicht mehr? Eine Klassifizierung von Rekursionsarten nach der Anzahl und Anordnung der rekursiven Aufrufe liefert Anhaltspunkte für diese Entscheidung.

Die Beispiele in diesem Abschnitt sind mathematisch orientiert und teilweise etwas praxisfern. Dafür zeigen sie aber die jeweils betrachtete Eigenschaft recht klar und lassen sich ohne viel Ballast in Java-Code umsetzen. Abschnitt 4.4 stellt realistischere Anwendungen der Rekursion vor.

4.3.1 Lineare Rekursion

Ein rekursiver Aufruf pro Aufruf

Bei **linearer Rekursion** zieht ein rekursiver Aufruf höchstens *einen einzigen* weite-

[12] Andere Programmiersprachen regeln den Umgang mit Endrekursion ausdrücklich. Beispielsweise müssen Scheme-Systeme Endrekursion immer eliminieren, Scala-Compiler unter bestimmten Voraussetzungen.

[13] Der alternative Java-Compiler „Jikes“ von IBM eliminierte Endrekursion. IBM entwickelt Jikes aber seit Java-Version 5 nicht mehr weiter.

ren direkten Aufruf nach sich. Die Rekursionstiefe, das heißt die maximale Anzahl der Stackframes, die geichzeitig auf dem Laufzeitstack gestapelt werden müssen, stimmt mit der Gesamtzahl der rekursiven Aufrufe überein. Beispiele für lineare Rekursion sind `Sum` (Listing 4.7) und `EvenOdd` (Listing 4.8). Endrekursion ist ein besonders einfacher Sonderfall linearer Rekursion, bei dem der rekursive Aufruf die letzte Aktion der Methode ist. Direkt nach dem rekursiven Aufruf kehrt die Methode zurück.

Umformung in Iterationen

Linear-rekursive Methoden lassen sich oft mit wenig Aufwand in eine iterative Fassung bringen. Ein eigener Stack verwaltet dazu die Informationen auf dem Laufzeitstack.

Simulation des Laufzeitstacks

Als Beispiel dient die weiter vorne eingeführte linear-rekursive Summenmethode:

```
int sum(final int n) {
    if(n > 0)
        return n + sum(n - 1);
    return 0;
}
```

Listing 4.17: Rekursive Methode zur Berechnung der Zahlensumme.

Der Methodenrumpf wird in zwei Teile zerlegt:

Aufteilen der rekursiven Methode

- Der erste Abschnitt (A) reicht bis zum rekursiven Aufruf, einschließlich der Berechnung der Argumente. Im Beispiel ist das nur eines, nämlich der Ausdruck `n - 1`.
- Der zweite Abschnitt (B) umfasst den Rest nach dem rekursiven Aufruf. Im Beispiel ist das die Addition von `n` zum Ergebnis der Rekursion.

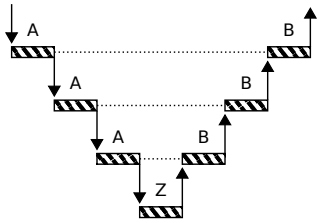
Dazu kommt der Codeabschnitt am Rekursionsende (Z), im Beispiel das Ergebnis `0`. Besser erkennt man diese Abschnitte, wenn der Code der Methode `sum` in etwas umständlicheren Einzelschritten ausformuliert ist:

```
int sum(final int n) {
    if(n > 0) {
        int temp = n - 1;           // A
        int result = sum(temp);
        result = result + n;        // B
        return result;
    }
```

```
    else
        return 0;                       // Z
}
```

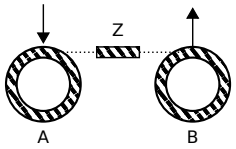
Listing 4.18: Rekursive Berechnung der Zahlensumme in Einzelschritten.

Das folgende Schema zeigt den Ablauf eines rekursiven Aufrufs:



Wiederholen der Methodenteile mit Schleifen

Zuerst wird A wiederholt, bis die Abbruchbedingung erfüllt ist. Dann folgt einmal Z und schließlich ebenso oft B wie vorher A. Die iterative Fassung verwendet passende Schleifen, um A und B zu wiederholen:



Die erste Schleife führt Codeabschnitt A aus. Sie sichert die Parameter vor der Berechnung der Argumente auf dem Stack.

```
int sum(int n) {
```

```
Stack<Integer> stack = new Stack<>();
while(n > 0) {
    stack.push(n);      // n sichern
    int temp = n - 1;
    n = temp;
}
// ...
```

Listing 4.19: Simulation rekursiver Aufrufe mit Sichern der Argumente auf einem Stack.

Eine zweite Schleife arbeitet, bis der Stack leer ist. Jeder Schleifendurchgang restauriert zunächst die Parameter vom Stack und wickelt dann die verbleibende Berechnung B ab, die hier nur aus der Aktualisierung des Ergebnisses besteht:

```
    // ...
    int result = 0;
    while(!stack.isEmpty()) {
        n = stack.pop();        // n restaurieren
        result += n;
    }
    return result;
}
```

Listing 4.20: Simulation der Rückkehr aus der Rekursion mit Abbau des Stacks.

Komplizierterer Code, aber kein Stacküberlauf

Nach dem gleichen Schema können alle linear-rekursiven Methoden in eine iterative Fassung gebracht werden. Der Preis für die beseitigte Gefahr des Laufzeitstacküberlaufs ist deutlich komplizierterer Code, der zudem die eigentliche Logik der Methode mit der Verwaltung des nachgebildeten Laufzeitstacks vermischt.[14]

Laufzeitstack überflüssig bei Endrekursion

Bei Endrekursion gibt es keinen Abschnitt B. Daher entfällt im oben gezeigten Umformungsschema die zweite Schleife. Nachdem keine Parameter mehr restauriert werden müssen, ist auch das Sichern in der ersten Schleife unnötig. Infolgedessen kann der ganze simulierte Stack entfallen.

4.3.2 Verzweigte Rekursion

Mehrere rekursive Aufrufe pro Aufruf

Anders als bei linearer Rekursion mündet bei verzweigter Rekursion ein rekursiver Aufruf in *zwei oder mehr* weitere Aufrufe. Das bedeutet, dass die Gesamtzahl der rekursiven Aufrufe exponentiell mit der maximalen Rekursionstiefe steigt.

[14] Auch der aus Objekten nachgebildete Stack könnte überlaufen. Er „lebt“ allerdings auf dem Heap, der in einer normalen JVM weit mehr Platz bietet als der Laufzeitstack. Die Gefahr des Überlaufs ist entsprechend geringer.

Beispiel: Fibonaccizahlen

Ein typisches Beispiel verzweigter Rekursion liefert die Definition der Fibonaccizahlen $f(n)$:

- Die ersten beiden Fibonaccizahlen liegen fest als $f(1) = 1$ und $f(2) = 1$.
- Für $n > 2$ ist $f(n) = f(n-1) + f(n-2)$, also die Summe der beiden vorhergehenden Fibonaccizahlen.

Das folgende Programm setzt diese Definition direkt um. `main` gibt einige Elemente der Folge aus:

```
public class Fibonacci {
    public long fib(int n) {
        if(n <= 2)
            return 1;
        return fib(n - 1) + fib(n - 2);
    }

    public static void main(String... args) {
        Fibonacci fibonacci = new Fibonacci();
        for(int n = 1; n < Integer.parseInt(args[0]); n++)
            System.out.printf("fib(%d) = %d%n", n, fibonacci.fib(n));
    }
}
```

Listing 4.21: Verzweigte Rekursion zur Berechnung der Fibonaccizahlen.

Der Programmstart liefert die ersten Fibonaccizahlen:

```
$ java Fibonacci 10
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
```

Ab etwa vierzig Elementen bremst das Programm spürbar ab. Dabei spielt die Hardware keine allzu große Rolle.

Messung der Laufzeit und der Anzahl rekursiver Aufrufe

Die folgende von `Fibonacci` (Listing 4.21) abgeleitete Klasse zählt die Anzahl der

rekursiven Methodenaufrufe in der Objektvariablen `calls` mit. Weiter hält sie die Dauer der Berechnung fest.[15]

```
public class FibonacciInstrumented extends Fibonacci {
    private long calls;

    private final long startMillis = System.currentTimeMillis();

    public long fib(int n) {
        calls++;
        return super.fib(n);
    }

    public String toString() {
        return "millis = " + (System.currentTimeMillis() - startMillis) + ", calls = " +
    }

    public static void main(String... args) {
        for(int n = 1; n < Integer.parseInt(args[0]); n++) {
            Fibonacci fibonacci = new FibonacciInstrumented();
            System.out.printf("fib(%d) = %d, %s%n", n, fibonacci.fib(n), fibonacci);
        }
    }
}
```

Listing 4.22: Berechnung der Fibonaccizahlen mit Protokoll der Aufrufe.

Ein Start des instrumentierten Programms bringt ans Licht, dass die Anzahl der rekursiven Aufrufe und die Laufzeiten selbst eine Art Fibonaccifolge bilden.[16]

Hohe Anzahl rekursiver Aufrufe

```
$ java FibonacciInstrumented 50
fib(1) = 1, millis = 9, calls = 1
fib(2) = 1, millis = 0, calls = 1
fib(3) = 2, millis = 0, calls = 3
fib(4) = 3, millis = 0, calls = 5
fib(5) = 5, millis = 0, calls = 9
...
fib(45) = 1134903170, millis = 31899, calls = 2269806339
fib(46) = 1836311903, millis = 52024, calls = 3672623805
fib(47) = 2971215073, millis = 83607, calls = 5942430145
fib(48) = 4807526976, millis = 136478, calls = 9615053951
fib(49) = 7778742049, millis = 221464, calls = 15557484097
```

[15] Diese simple Laufzeitmessung liefert erst bei Zeitspannen von einigen Sekunden halbwegs reproduzierbare Werte und ist für kürzere Messungen schlecht geeignet. Das Betriebssystem, die JVM und andere Programme sind oft mit anderen Aufgaben beschäftigt, wodurch kurze Zeitintervalle stark verfälscht werden können.

[16] Das ist wenig überraschend: Um $f(n)$ zu berechnen sind die Aufrufe für $f(n-1)$ nötig, dazu die Aufrufe für $f(n-2)$, insgesamt also die Summe der Aufrufanzahlen, zuzüglich eines Aufrufs für $f(n)$ selbst. Unter der Annahme, dass jeder Aufruf ungefähr gleich lang dauert, ist die Laufzeit proportional zur Anzahl der Aufrufe.

Umformung in Iteration

Rentabilität der Simulation des Laufzeitstacks

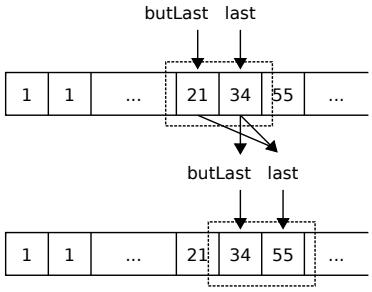
Verzweigte Rekursion lässt sich nicht so leicht in eine iterative Fassung bringen wie lineare Rekursion. In den Stackframes steckt zusätzlich zur Aufrufumgebung auch der wechselnde Ort des Aufrufs. Diese Information müsste eine iterative Methode explizit selbst verwalten. Die dazu nötigen Datenstrukturen entsprechen letztlich dem Laufzeitstack. Der Code zur Verwaltung dieser Datenstrukturen kommt zur Lösung des eigentlichen Problems dazu und bläht diese zusätzlich auf. Ob der eigene Code dann effizienter als die JVM arbeitet, ist fraglich.

Vorwärtsberechnung der Fibonaccizahlen

Schnelle Lösung speziell für Fibonaccizahlen

Fibonaccizahlen lassen sich einfacher berechnen: Eine Zahl ergibt sich alleine aus den beiden vorhergehenden Zahlen. Weiter zurückliegende Zahlen spielen keine Rolle und brauchen nicht gespeichert zu werden.

Die folgende iterative Fassung hält die jeweils letzte und vorletzte Fibonaccizahl in den lokalen Variablen `last` und `butLast` fest. Jeder Schleifendurchgang ersetzt sie durch die beiden nächsten Werte, bis die gewünschte Anzahl Werte berechnet ist. Bildlich betrachtet wird eine Art „Fenster“ über die Fibonaccifolge geschoben, in dem immer die beiden zuletzt berechneten Werte sichtbar sind:



Jeder Schleifendurchgang aktualisiert die Variablen `last` und `butLast`. Der vorher letzte Wert wird dabei zum neuen vorletzten. Der neue letzte Wert ergibt sich aus der Summe der beiden alten Werte:

```
butLast = last;
last = last + butLast;
```

Allerdings funkioniert die Umsetzung der Idee so nicht: Die beiden Variablen müssten *gleichzeitig* aktualisiert werden. Die beiden Wertzuweisungen laufen dagegen zeitlich *nacheinander* ab. Sie können in keine korrekte Reihenfolge gebracht werden. Die folgende Methode löst das Problem mit einer Hilfsvariablen `temp`:

```
public long fib(int n) {
    long last = 1;
    long butLast = 1;
    while(n > 2) {
        long temp = last;
        last = last + butLast;
        butLast = temp;
        n--;
    }
    return last;
}
```

Listing 4.23: Iterative Berechnung von Fibonaccizahlen.

Die Methode arbeitet sehr schnell und liefert die ersten 50 Werte praktisch sofort. Allerdings nutzt sie eine spezielle Eigenschaft der Fibonaccifolge und kann nicht auf andere Fälle verzweigter Rekursion übertragen werden.

4.3.3 Geschachtelte Rekursion

Mehrere sequenzielle rekursive Aufrufe pro Aufruf

Bei geschachtelter Rekursion erfordert wenigstens ein *Argument* eines rekursiven Aufrufs einen weiteren rekursiven Aufruf. Das bedeutet, dass ein vorhergehender Aufruf abgeschlossen sein muss, bevor ein weiterer überhaupt gestartet werden kann. Diese Eigenschaft verhindert parallele Verarbeitung und zwingt zur sequenziellen Berechnung.

Beispiel: Ackermann-Funktion

Ein Beispiel geschachtelter Rekursion ist die „Ackermann-Funktion“, die einerseits interessante Eigenschaften für die theoretische Informatik aufweist, andererseits auch als Belastungstest für Funktionsaufrufe im Laufzeitsystem dient. Diese schnell wachsende Funktion ack hängt von zwei Parametern ab und ist folgendermaßen definiert:[17]

- $ack(0, n) = n + 1$
- $ack(m, 0) = ack(m - 1, 1)$
- $ack(m, n) = ack(m - 1, ack(m, n - 1))$ für $m > 0$ und $n > 0$.

Sie liefert schon für kleine Argumente sehr große Werte. Das Programm `Ackermann` implementiert diese Definition:

[17] Die Ackermann-Funktion ist für alle nicht negativen Argumente definiert, auch wenn das dieser Definition nicht ohne Weiteres anzusehen ist.

```
public class Ackermann {
    public long ack(long m, long n) {
        if(m == 0)
            return n + 1;
        if(n == 0)
            return ack(m - 1, 1);
        return ack(m - 1, ack(m, n - 1));
    }

    public static void main(String... args) {
        int m = Integer.parseInt(args[0]);
        int n = Integer.parseInt(args[1]);
        System.out.println(new Ackermann().ack(m, n));
    }
}
```

Listing 4.24: Geschachtelte Rekursion zur Berechnung der Ackermann-Funktion.

Hohe Anzahl rekursiver Aufrufe schon bei kleinen Argumentwerten

Das Programm liefert die Werte für einige kleine Argumente:[18]

```
$ java Ackermann 3 3
61
$ java Ackermann 3 4
125
$ java -Xss4M Ackermann 4 1
65533
```

Rasch steigende Ergebniswerte

Schon zur Berechnung von $ack(4, 2)$ reichen die primitiven Typen von Java nicht mehr aus. Das Ergebnis hat etwa 20000 Dezimalstellen.

Eine instrumentierte Fassung des Programms protokolliert die Laufzeit, die Anzahl der Aufrufe und die maximale Rekursionstiefe:

```
$ java AckermannInstrumented 3 3
ack(3, 3) = 61, millis = 10, calls = 2432, max depth = 63
$ java AckermannInstrumented 3 4
ack(3, 4) = 125, millis = 27, calls = 10307, max depth = 127
$ java -Xss4M AckermannInstrumented 4 1
ack(4, 1) = 65533, millis = 105040, calls = 2862984010, max depth = 65536
```

Beispiel: Takeuchi-Funktion

Ein anderes Beispiel für geschachtelte Rekursion ist die „Takeuchi-Funktion“ tak:

[18] Das Programm kann $ack(4, 1)$ noch berechnen. Allerdings erfordert die hohe Rekursionstiefe einen größeren Laufzeitstack als die JVM normalerweise zur Verfügung stellt. Der Kommandozeilenschalter `-Xss`*size* (Seite 246) fordert einen Laufzeitstack der Größe *size* an, in diesem Beispiel 4 Megabyte.

- $tak(x, y, z) = z$ für $x \leq y$ und
- $tak(x, y, z) = tak(tak(x - 1, y, z), tak(y - 1, z, x), tak(z - 1, x, y))$ ansonsten.

Die Definition lässt sich direkt in Java-Code übertragen:

```
public class Takeuchi {
    public long tak(long x, long y, long z) {
        if(x <= y)
            return z;
        return tak(tak(x - 1, y, z), tak(y - 1, z, x), tak(z - 1, x, y));
    }
}
```

Listing 4.25: Geschachtelte Rekursion mit geringer maximaler Rekursionstiefe.

Viele rekursive Aufrufe bei geringer Rekursionstiefe

Die Takeuchi-Funktion erfordert eine große Anzahl rekursiver Aufrufe, erreicht dabei aber keine große Tiefe und liefert keine großen Ergebniswerte, wie eine instrumentierte Fassung des Programms zeigt:[19]

```
$ java TakeuchiInstrumented 18 12 6
tak(18, 12, 6) = 7, millis = 39, calls = 63609, max depth = 18
```

Umformung in Iteration

Umformung in iterative Fassung kaum sinnvoll

Eine geschachtelt-rekursive Methode kann nur schwer in eine iterative Fassung umgeschrieben werden. Obwohl es prinzipiell immer möglich wäre, lohnt sich der Aufwand in der Praxis nicht. Die Performance-Nachteile einer rekursiven Lösung können in manchen Fällen durch „Memoizing“ (Seite 277) ausgelichen werden.

4.4 Anwendungen

4.4.1 Filesystem

Prototyp einer rekursiven Datenstruktur

Ein Filesystem ist ein typisches Beispiel einer rekursiven Datenstruktur. Operationen im Filesystem lassen sich daher gut mit rekursiven Methoden umsetzen. Die Java-Bibliothek bietet dabei Unterstützung mit der Methode `walkFileTree` der Klasse `Files` (Seite 106). Ungeachtet der vordefinierten Bibliotheksmethode soll eine vergleichbare Funktion hier neu entwickelt werden.

[19] Im Gegensatz zur Ackermann-Funktion ist es bei der Takeuchi-Funktion nicht nötig, die Stackgröße explizit zu erhöhen.

Rekursiver Durchlauf

Durchsuchen eines Directorybaums

Die Methode `walk` im folgenden Programm `SimpleDirWalker` erwartet als Argument ein Element eines Filesystems als `Path`-Objekt. In einem modernen Filesystem gibt es viele verschiedene Bewohner. `walk` interessiert sich davon nur für reguläre Files und Directories und ignoriert alles andere.

- Reguläre Files werden ausgegeben.
- Bei Directories liefert ein `DirectoryStream` die einzelnen Elemente. Mit jedem Element wird die Methode `walk` rekursiv aufgerufen.

```
import java.io.*;
import java.nio.file.*;

public class SimpleDirWalker {
    void walk(Path path) throws IOException {
        if(Files.isRegularFile(path))
            System.out.println(path);
        else if(Files.isDirectory(path))
            for(Path element: Files.newDirectoryStream(path))
                walk(element);
    }
}
```

Listing 4.26: Directorybaum rekursiv durchlaufen und Files ausgeben.

Eine `main`-Methode startet den Filesystem-Durchlauf mit dem Directory, das das Kommandozeilenargument vorgibt.

```
public static void main(String... args) throws IOException {
    new SimpleDirWalker().walk(Paths.get(args[0]));
}
```

Listing 4.27: Start des rekursiven Directory-Durchlaufes.

Das Programm listet korrekt die Files auf:

```
$ java SimpleDirWalker .
./SimpleDirWalker.class
./PathProcessor.class
./DirWalker$1.class
./DirWalker.class
./unused/HalfFib.class
./unused/FibTail.class
```

Man kann sich viele andere Operationen mit Files und Directories vorstellen als nur das Auflisten. Der eigentliche Durchlauf bleibt dabei immer gleich. Die Zeilenkommentare in der obigen Methode `walk` zeigen die Stellen, an denen Code zur Verarbeitung einzufügen wäre.

Trennung Durchlauf und Verarbeitung

Abstraktion der Verarbeitung von Filesystem-Elementen

Im Sinne der Modularisierung sollte der Durchlauf des Directorybaums von der Verarbeitung der Bewohner getrennt bleiben. Die Verarbeitung wird deshalb in Klassen ausgelagert, die der Anwender nach eigenen Vorstellungen definieren kann. Die einzige Forderung ist Kompatibilität zum Interface `PathProcessor` mit der Methode `accept`:

```
import java.io.*;
import java.nio.file.*;

public interface PathProcessor {
    void accept(Path path) throws IOException;
}
```

Listing 4.28: Interface für Klassen zur Verarbeitung von Filesystem-Elementen.

Den Filesystem-Durchlauf übernimmt die Klasse `DirWalker`. Ein `DirWalker` referenziert vier `PathProcessor`-Objekte:

- `at` für reguläre Files,
- `down` beim Betreten von Directories,
- `up` beim Verlassen von Directories und
- `alien` für andere Elemente als reguläre Files und Directories.

Zur Vereinfachung darf jedes der Objekte fehlen, das heißt `null` sein. Andernfalls ruft `walk` beim Durchlauf die `accept`-Methode des betreffenden Objekts auf.

```
import java.io.*;
import java.nio.file.*;

public class DirWalker {
    private final PathProcessor at;

    private final PathProcessor down;

    private final PathProcessor up;

    private final PathProcessor alien;
```

```
    public DirWalker(PathProcessor at, PathProcessor down, PathProcessor up, PathProcesso
        this.at = at;
        this.down = down;
        this.up = up;
        this.alien = alien;
    }

    public void walk(Path path) throws IOException {
        if(Files.isRegularFile(path)) {
            if(at != null)
                at.accept(path);
        }
        else if(Files.isDirectory(path)) {
            if(down != null)
                down.accept(path);
            for(Path element: Files.newDirectoryStream(path))
                walk(element);
            if(up != null)
                up.accept(path);
        }
        else if(alien != null)
            alien.accept(path);
    }
}
```

Listing 4.29: Rekursiver Durchlauf eines Directorybaumes mit Aufruf von Callbacks.

Anwendungen des Filesystem-Durchlaufs

Ein Anwender der Klasse `DirWalker` stellt `PathProcessor`en seiner Wahl zur Verfügung und startet dann den Durchlauf eines Directorybaums. Das folgende Beispiel gibt jedes File aus und ignoriert den Rest. Es entspricht der oben gezeigten Klasse `SimpleDirWalker`:

```
import java.io.*;
import java.nio.file.*;

public class PrintFiletree {
    public static void main(String... args) throws IOException {
        PathProcessor printer = new PathProcessor() {
            public void accept(Path path) {
                System.out.println(path);
            }
        };
        new DirWalker(printer, null, null, null).walk(Paths.get(args[0]));
    }
}
```

Listing 4.30: Anwendung des rekursiven Directory-Durchlaufes: Auflisten der Dateien.

Löschen leerer Directories

Eine andere Anwendung des `DirWalker` löscht Directories, die keine Files enthalten. Der `PathProcessor` zündet beim Verlassen von Directories. Zu diesem Zeitpunkt sind eventuell enthaltene Subdirectories, die ihrerseits leer waren, bereits abgeräumt, sodass ganze Directorybäume ohne Inhalt komplett gelöscht werden.

```
import java.io.*;
import java.nio.file.*;

public class EmptyDirClipper {
    public static void main(String... args) throws IOException {
        PathProcessor emptyDirClipper = new PathProcessor() {
            public void accept(Path path) throws IOException {
                if(!Files.newDirectoryStream(path).iterator().hasNext()) {
                    System.out.println("deleting " + path);
                    Files.delete(path);
                }
            }
        };
        new DirWalker(null, null, emptyDirClipper, null).walk(Paths.get(args[0]));
    }
}
```

Listing 4.31: Anwendung des rekursiven Directory-Durchlaufes: Löschen leerer Directories.

Doppelte Files

Das letzte Anwendungsbeispiel sucht nach potenziellen Duplikaten von Files. Dazu baut der `PathProcessor` die Map `checksums2path` auf, die Prüfsummen auf Files abbildet. Files mit derselben Prüfsumme sind Kandidaten für Duplikate und werden ausgegeben.

Die Berechnung der Prüfsumme schließt einen Kompromiss zwischen Geschwindigkeit und Verlässlichkeit: Sie liest nur das erste Kilobyte eines Files und berechnet daraus mithilfe der Filterklasse `CheckedInputStream` (siehe Seite 62) eine CRC32-Prüfsumme. [20]

```
import java.io.*;
import java.nio.file.*;
import java.util.*;
import java.util.zip.*;
```

[20] Die Erkennung von Duplikaten ist hier der Kürze wegen etwas grob implementiert. Als „billigen" Voraustest könnte man beispielsweise noch die Dateilängen vergleichen. Bei gleicher Prüfsumme des ersten Kilobytes sollte man sich auch den Rest der Datei-Inhalte vornehmen.

```
public class DupScanner {
    public static void main(String... args) throws IOException {
        PathProcessor dupScanner = new PathProcessor() {
            private final Map<Long, Path> checksums2path = new HashMap<>();

            private final byte[] buffer = new byte[1024];

            private long checksum(Path path) throws IOException {
                try(InputStream in = new FileInputStream(path.toString());
                    CheckedInputStream checked = new CheckedInputStream(in, new CRC32()))
                    checked.read(buffer);
                    return checked.getChecksum().getValue();
                }
            }

            public void accept(Path path) throws IOException {
                long checksum = checksum(path);
                if(checksums2path.containsKey(checksum))
                    System.out.printf("%s -- %s%n", path, checksums2path.get(checksum));
                else
                    checksums2path.put(checksum, path);
            }
        };
        new DirWalker(dupScanner, null, null, null).walk(Paths.get(args[0]));
    }
}
```

Listing 4.32: Anwendung des rekursiven Directory-Durchlaufes: Aufspüren von Datei-Duplikaten.

4.4.2 Parser

Formaler Text mit Klammerstrukturen

Ein Parser analysiert einen Text, der formalen Regeln folgt, und wertet die gefundenen Bestandteile aus. Oft baut der Parser Datenstrukturen auf und übergibt sie an den Rest des Programms, das damit weiterarbeitet. In früheren Kapiteln tauchten schon verschiedene Parser auf, zum Beispiel bei der Deserialisierung von Objekten (Kapitel 2) und beim Einlesen von XML-Dokumenten (Kapitel 3). Die Konstruktion von Parsern ist im Allgemeinen nicht ganz einfach und fällt in den Bereich des Compilerbaus.

Vereinfachte arithmetische Ausdrücke

Hier soll ein Parser implementiert werden, der „vollständig geklammerte arithmetische Ausdrücke“ (*Fully Parenthesized Expression*, FPE) analysiert und auswertet. Ein FPE ist ein arithmetischer Ausdruck im Sinne von Java, allerdings mit einigen Einschränkungen:[21]

- Eine einzelne Dezimalziffer ist ein elementarer Ausdruck. Der Wert dieses Ausdrucks ist der Zahlenwert der Ziffer.

[21] FPEs schließen alle Fragen der Priorität und Assoziativität von binären Operatoren aus, weil sie Klammern um die sich entsprechenden, zusammengesetzten Teilausdrücke verlangen.

- Ein zusammengesetzter Ausdruck besteht aus einem ersten Teilausdruck, einem Operatorzeichen und einem zweiten Teilausdruck. Diese Bestandteile müssen immer in runde Klammern eingefasst werden. Der Wert des ganzen Ausdrucks ist die arithmetische Verknüpfung der beiden Teilausdrücke.
- Ein Ausdruck kann mit einem vorangestellten Minuszeichen negiert werden.

Definition und Beispiele von FPEs

Die folgende Grammatik legt den Aufbau von FPEs genau fest. `F` ist ein FPE. `d` steht für eine Dezimalziffer, `p` für eines der Operatorzeichen `+`, `-`, `*` und `/`. Die Klammern und das Minuszeichen stehen für sich selbst.[22]

```
F → d
F → - F
F → ( F p F )
```

Hier ein paar Beispiele für FPEs mit ihren Werten.

```
4 ⇒ 4.0
-(2*4) ⇒ -8.0
((-3+5)/-(7--3)) ⇒ -0.2
----5 ⇒ 5.0
```

FPE-Parser liest zeichenweise

Der FPE-Parser arbeitet sich Zeichen für Zeichen durch die Eingabe. Sein Aufbau folgt der Grammatik. Er besteht im Kern aus der Methode `expression`, deren Aufruf einen kompletten FPE liest und dessen `double`-Wert zurückliefert. `expression` entscheidet sich anhand des aktuellen Eingabezeichens, das in der Objektvariablen `current` gespeichert ist, für eine der drei folgenden Möglichkeiten:

Dezimalziffer
: Der Wert des Ausdrucks ist der Zahlenwert der Ziffer. Die Ziffer wird verworfen und das nächste Eingabezeichen geholt.

Minuszeichen
: Das Minuszeichen wird verworfen und mit einem rekursiven Aufruf der nächste FPE gelesen, dessen Wert negiert wird.

Öffnende Klammer
: Die öffnende Klammer wird verworfen und mit einem rekursiven Aufruf ein weiterer FPE gelesen. Dann wird ein Operatorzeichen geholt und mit einem zweiten rekursiven Aufruf ein weiterer Ausdruck gelesen. Die

[22] Diese Schreibweise wird als „Backus-Naur-Form“ (BNF) bezeichnet. Es macht nichts, wenn Sie sie nicht gegenwärtig haben. Lesen Sie die Zeilen als Sammlung von Konstruktionsregeln, nach denen die linke Seite so aufgebaut sein darf, wie rechts vom Pfeil angegeben ist.

Werte der rekursiv gelesenen Teilausdrücke werden gemäß Operatorzeichen zum Gesamtwert verknüpft; danach wird noch eine schließende Klammer gelesen und verworfen.

Anderes Zeichen
: Ein korrekter Ausdruck kann mit keinem anderen Zeichen beginnen. Der Parser meldet einen Syntaxfehler und bricht mit einer Exception ab.

Rekursive Implementierung des Parsers

Hier der Code der rekursiven Methode. Die Objektvariable `current` speichert das jeweils aktuelle Eingabezeichen.

```
private int current;

double expression() throws SyntaxErrorException, IOException {
    double value;
    if(Character.isDigit(current))
        value = consume("0123456789") - '0';
    else if(current == '-') {
        consume("-");
        value = -expression();
    }
    else if(current == '(') {
        consume("(");
        value = expression();
        int operator = consume("+-*/");
        double second = expression();
        switch(operator) {
            case '+':
                value += second;
                break;
            case '-':
                value -= second;
                break;
            case '*':
                value *= second;
                break;
            case '/':
                value /= second;
                break;
        }
        consume(")");
    }
    else
        throw new SyntaxErrorException("expected one from \"(-0123456789\", got " + (char
    return value;
}
```

Listing 4.33: Rekursive Berechnung des Wertes eines arithmetischen Ausdrucks.

Erwartetes Einzelzeichen holen

Die Hilfsmethode `consume` erhält einen String mit einer Auswahl von akzeptablen Zeichen und stellt sicher, dass das aktuelle Zeichen darunter zu finden ist. Wenn

das das Fall ist, liefert die Methode das akzeptierte Zeichen zurück und holt ein neues aktuelles Zeichen von der Eingabe. Zwischenraum lässt sie dabei aus.

```
private Reader input;

private int consume(String choice) throws SyntaxErrorException, IOException {
    int result = current;
    if(choice.indexOf(current) >= 0)
        do  // skip whitespace
            current = input.read();
        while(Character.isWhitespace(current));
    else
        throw new SyntaxErrorException("expected one from \"" + choice + "\", got " + (ch
    return result;
}
```

Listing 4.34: Konsumieren von Eingabezeichen.

Der Konstruktor des Parsers erwartet einen Reader und sorgt dafür, dass das erste Zeichen verfügbar ist:

Initialisierung und Start des Parsers

```
import java.io.*;

public class FPEParser {
    // ...
    public FPEParser(Reader input) throws IOException {
        this.input = input;
        current = input.read();
    }
}
```

Listing 4.35: Konstruktor für den Parser.

Eine `main`-Methode übergibt das Kommandozeilenargument als Eingabe an den Parser und druckt das Ergebnis aus:

```
public static void main(String... args) throws SyntaxErrorException, IOException {
    try(Reader input = new StringReader(args[0])) {
        System.out.println(new FPEParser(input).expression());
    }
}
```

Listing 4.36: Start des Parsers mit einem arithmethischen Ausdruck von der Kommandozeile.

Der Parser funktioniert wie erwartet:[23]

Ablaufbeispiele des Parsers

[23] Die Gänsefüßchen sind nötig um die Klammern unbehelligt an der Unix-Shell vorbei zum Java-Programm zu bringen.

```
$ java FPEParser 4
4.0
$ java FPEParser "-(2 * 4)"
-8.0
$ java FPEParser "((-3 + 5) / -(7 -- 3))"
-0.2
$ java FPEParser ----5
5.0
```

Falsche Eingaben beanstandet der Parser mit einer annehmbaren Fehlermeldung:[24]

```
$ java FPEParser -+
Exception in thread "main" parser.SyntaxErrorException:
    expected one of "(-0123456789", got +
$ java FPEParser "((-3+5)/-()"
Exception in thread "main" parser.SyntaxErrorException:
    expected one of "(-0123456789", got )
```

Recursive Descent Parser

Die Bauart des hier entwickelten Parsers wird im Compilerbau als *Recursive Descent Parser* bezeichnet. Solche Parser lassen sich mechanisch aus einer Grammatik generieren, wenn sie gewisse Voraussetzungen erfüllt. Gemeinsam ist all diesen Parsern die rekursive Arbeitsweise.

4.4.3 Permutationen

Anordnungen von n verschiedenen Objekten

Als Permutation von n verschiedenen Objekten bezeichnet man eine Anordnung in einer Reihe. Die Anzahl *aller möglichen* Permutationen ist das Produkt $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 = n!$, die „Fakultät von n“. Für die Zahlen 1, 2, 3 gibt es beispielsweise $3! = 3 \cdot 2 \cdot 1 = 6$ Permutationen:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Rekursive Methode zum Berechnen von Permutationen

Hier wird eine Methode `generate` entwickelt, die alle Permutationen einer Menge von Objekten berechnet und verarbeitet. `generate` erwartet zwei Parameter:

1. eine halbfertige Permutation als Liste von Objekten und

[24] Der abgedruckte Stacktrace ist gekürzt.

2. die Menge der restlichen Objekte, die in der halbfertigen Permutation noch fehlen.

`generate` verschiebt rekursiv ein Objekt nach dem anderen von der Restmenge in die Permutation.

- Wenn die Restmenge leer ist, sind alle Objekte in die Permutation eingefügt. Damit ist eine Permutation komplett und wird zur Kontrolle ausgegeben. Daraus ergibt sich die Abbruchbedingung für die Rekursion.
- Andernfalls nimmt die Methode in einer Schleife ein Objekt nach dem anderen aus der Restmenge und fügt es hinten an die halbfertige Permutation an. Mit einem rekursiven Aufruf werden dann die verbleibenden Objekte verarbeitet.

Generische Methode

Für `generate` ist der Typ der Objekte unerheblich. Die Methode kann daher generisch bezüglich des Objekttyps bleiben:

```
static <T> void generate(List<T> permutation, Set<T> rest) {
    if(rest.isEmpty())
        System.out.println(permutation);
    else
        for(T element: rest) {
            permutation.add(element);
            rest.remove(element);
            generate(permutation, rest);
            rest.add(element);
            permutation.remove(element);
        }
}
```

Listing 4.37: Berechnung von Permutationen, scheitert mit `ConcurrentModificationException`.

Fehler wegen falscher Verwendung einer Collection

Die folgende `main`-Methode versucht, mit `generate` die Permutationen der Kommandozeilenargumente zu generieren. Als Argumente erhält `generate` eine leere Liste (die noch leere Permutation) und eine Menge von Strings, die mit den Kommandozeilenargumenten initialisiert wird:

```
public static void main(String... args) {
    Set<String> elements = new HashSet<>(Arrays.asList(args));
    generate(new ArrayList<String>(), elements);
}
```

Listing 4.38: Berechnung der Permutationen der Kommandozeilenargumente.

Der Aufruf des Programm stürzt allerdings mit einer Exception ab:

```
$ java Permutations 1 2 3
[3, 2, 1]
Exception in thread "main" java.util.ConcurrentModificationException
```

Das Problem ist die *foreach*-Schleife in `generate`: Sie iteriert über die Menge `rest`, die im Rumpf der Schleife mit den Aufrufen von `remove` und `add` verändert wird. Die Änderungen entwerten den Iterator, den die *foreach*-Schleife implizit benutzt.

Korrektur auf Kosten von Ressourcenverbrauch

Als Ausweg kann die Restmenge im Schleifenrumpf in eine neue Menge kopiert werden, die anstelle der Originalmenge verwendet und ohne Schaden verändert werden kann:

```
static <T> void generate(List<T> permutation, Set<T> rest) {
    if(rest.isEmpty())
        ;
    else
        for(T element: rest) {
            permutation.add(element);
            Set<T> restCopy = new HashSet<>(rest);
            restCopy.remove(element);
            generate(permutation, restCopy);
            permutation.remove(element);
        }
}
```

Listing 4.39: Berechnung von Permutationen mit Kopieren der Menge der Restelemente.

Jetzt arbeitet die Methode fehlerfrei:

```
$ java Permutations 1 2 3
[3, 2, 1]
[3, 1, 2]
[2, 3, 1]
[2, 1, 3]
[1, 3, 2]
[1, 2, 3]
```

Verarbeitung der generierten Permutationen

In der bisher entwickelten Fassung gibt `generate` jede Permutation aus. Das ist anschaulich, aber nicht besonders flexibel. Besser sollte es dem Anwender der Klasse selbst überlassen werden, wie er die generierten Permutationen verwenden möchte.

Dazu erhält `generate` einen weiteren Parameter für ein Objekt mit einer Callback-Methode, die mit jeder generierten Permutation aufgerufen wird. Ein generisches Interface `PermutationEater` stellt die Kompatibilität der Objekte sicher und definiert die Schnittstelle der Callback-Methode:

```
import java.util.*;

public interface PermutationEater<T> {
    void eat(List<T> permutation);
}
```

Listing 4.40: Interface von Klassen zur Verarbeitung von Permutationen.

`generate` wird um den Parameter `eater` und den Aufruf der Callback-Methode erweitert:

```
import java.util.*;

public class PermutationGenerator {
    public static <T> void generate(List<T> permutation, Set<T> rest, PermutationEater<T>
        if(rest.isEmpty())
            eater.eat(permutation);
        else
            for(T element: rest) {
                permutation.add(element);
                Set<T> restCopy = new HashSet<>(rest);
                restCopy.remove(element);
                generate(permutation, restCopy, eater);
                permutation.remove(element);
            }
    }
}
```

Listing 4.41: Generator von Permutationen mit Weitergabe an einen `PermutationEater`.

Ausgeben der Permutationen

Die folgende `main`-Methode erzeugt einen `PermutationEater` namens `printer`, der die Permutationen ausgibt.

```
import java.util.*;

public class PermutationPrinter {
    public static void main(String... args) {
        Set<String> elements = new HashSet<>(Arrays.asList(args));
        PermutationEater<String> printer = new PermutationEater<String>() {
            public void eat(List<String> permutation) {
                System.out.println(permutation);
            }
        };
        PermutationGenerator.generate(new ArrayList<String>(), elements, printer);
    }
}
```

Listing 4.42: Ausgabe einer Liste von Permutationen.

Das Programm produziert die gleiche Ausgabe wie die vorhergehende `main`-Methode:

```
$ java PermutationPrinter 1 2 3
[3, 2, 1]
[3, 1, 2]
[2, 3, 1]
[2, 1, 3]
[1, 3, 2]
[1, 2, 3]
```

Abzählen der Permutationen

Zur Messung der Performance eignet sich der `printer` nicht gut, weil alleine die Ausgabe vergleichsweise viel Zeit kostet und die Laufzeitmessung verfälscht. Ein neuer `PermutationEater` zählt die Anzahl der Permutationen. So kann nachher überprüft werden, ob tatsächlich die korrekte Anzahl Permutationen generiert wurde:

```
import java.util.*;

public class PermutationCounterBroken {
    public static void main(String... args) {
        Set<String> elements = new HashSet<>(Arrays.asList(args));
        PermutationEater<String> counter = new PermutationEater<String>() {
            private int calls = 0;

            public void eat(List<String> permutation) {
                calls++;
            }

            public int getCalls() {
                return calls;
            }
        };
        PermutationGenerator.generate(new ArrayList<String>(), elements, counter);
        System.out.println(counter.getCalls()); // wird nicht übersetzt!
    }
}
```

Listing 4.43: Anonyme Klasse mit einer zusätzlichen nicht aufrufbaren Methode.

Anonyme Klasse mit nicht aufrufbaren zusätzlichen Methoden

An dieser Stelle taucht eine unerwartete Hürde auf. Der Ausdruck

```
counter.getCount()
```

in der letzten Ausgabeanweisung wird nicht übersetzt, weil `PermutationEater`, der statische Typ von `counter`, die Methode `getCount` nicht kennt! Auch ein Typecast ist nicht möglich, weil der wahre dynamische Typ von `counter` anonym ist und damit überhaupt keinen Namen hat.[25]

[25] Tatsächlich hat die anonyme Klasse einen Namen, der sich aus dem Namen der Toplevel-Klasse, hier `PermutationCounterBroken`, einem $-Zeichen und einer laufenden Nummer zusammensetzt. Nur hilft diese Erkenntnis nicht weiter, weil die Nummer vom Compiler nach Bedarf vergeben und nicht verlässlich ist.

Eine lokale Klassendefinition löst dieses Problem. Diese Art der Klassendefinition steht auf der Ebene von Anweisungen im Rumpf einer Methode und gilt für diesen einen Methodenrumpf:

Lösung mit einer lokalen Klasse

```
import java.util.*;

public class PermutationCounter {
    public static void main(String... args) {
        Set<String> elements = new HashSet<>(Arrays.asList(args));
        class PermutationCounterEater<T> implements PermutationEater<T> {
            private int calls = 0;

            public void eat(List<T> permutation) {
                calls++;
            }

            public int getCalls() {
                return calls;
            }
        };
        PermutationCounterEater<String> counter = new PermutationCounterEater<>();

        System.out.println(counter.getCalls());

    }
}
```

Listing 4.44: Lokale Klasse zum Mitzählen von generierten Permutationen.

Ein Programmaufruf zeigt, dass die erwartete Anzahl Permutationen berechnet wird:

```
$ java PermutationCounter 1 2 3
6
$ java PermutationCounter 1 2 3 4 5 6 7 8 9 10 11
39916800
```

Das Generieren der fast 40 Millionen Permutationen im zweiten Aufruf kostet einige Zeit, wie eine instrumentierte Fassung des Programms nachweist:[26]

Schlechte Laufzeit

```
$ java PermutationCounter 1 2 3 4 5 6 7 8 9 10 11
39916800
136475 millis
```

[26] Die Anzahl Millisekunden kann nur als relatives Maß dienen und ist nicht auf andere Systeme übertragbar. Hier wurde ein Netbook mit einer Intel-Atom-N270-CPU mit 1,6 GHz Taktfrequenz verwendet.

Ersetzen in einer Liste statt Kürzen und Anfügen

Mit ein paar Änderungen lässt sich die Methode `generate` deutlich beschleunigen, ohne dabei die gut nachvollziehbare rekursive Arbeitsweise aufzugeben.

1. Das Verlängern und Kürzen der halbfertigen Permutation `permutation` in der Schleife ist kostspielig, weil die Liste dabei ständig wächst und schrumpft, das heißt, ihre Struktur ändert:

```
for(T element: rest) {
   permutation.add(element);
   ...
   permutation.remove(element);
}
```

Besser wird gleich beim ersten Aufruf eine Liste ausreichender Länge erzeugt und Elemente mit `set` eingetragen. Ein zusätzlicher Zähler `length` führt Buch, wie viele Elemente der Liste gültige Daten enthalten. Der Rest ist unbenutzt. Die Liste ändert jetzt zwar ihren Inhalt, aber nicht mehr die Struktur.
Aus der halbfertigen Permutation muss nichts mehr gelöscht werden, weil das jeweils letzte Element in der Schleife laufend überschrieben wird:

```
import java.util.*;

public class FasterPermutations {
    static <T> void generate(int length, List<T> permutation, Set<T> rest, Permutation
        if(length >= permutation.size())
            eater.eat(permutation);
        else
            for(T element: rest) {
                permutation.set(length, element);
                Set<T> restCopy = new HashSet<>(rest);
                restCopy.remove(element);
                generate(length + 1, permutation, restCopy, eater);
            }
    }
}
```

Listing 4.45: Schnelleres Generieren von Permutationen ohne Strukturänderung der Elementliste.

Vermeiden von Objektkopien

2. Besonders „teuer“ ist das Umkopieren der restlichen Elemente in das neue temporäre Set `restCopy` in der Schleife:

```
for(T element: rest) {
   ...
   Set<T> restCopy = new HashSet<>(rest);
   ...
}
```

Schnellere Buchführung der enthaltenen Elemente

Letztlich ist nur wichtig, ob ein bestimmtes Element noch frei oder schon in die Permutation eingebaut ist. Für diese Buchführung wird die Menge `rest` durch

eine unveränderliche Elementliste ersetzt und um ein gleich langes `boolean`-Array `used` ergänzt, das „parallel“ neben `rest` liegt. Die `boolean`-Elemente von `used` geben Auskunft, ob das korrespondierende Elemente in `rest` schon Teil der Permutation ist (`true`) oder nicht (`false`).
Eine einfache `boolean`-Zuweisung ersetzt das Ein- und Ausfügen von Elementen in `rest`. Die `generate`-Methode erzeugt jetzt überhaupt keine neuen Objekte mehr und lässt die beiden Listen strukturell unverändert.

```
import java.util.*;

public class FastPermutations {
    static <T> void generate(int length, List<T> permutation, boolean[] used, List<T>
        if(length >= used.length)
            eater.eat(permutation);
        else
            for(int n = 0; n < used.length; n++) {
                if(used[n])
                    continue;
                permutation.set(length, elements.get(n));
                used[n] = true;
                generate(length + 1, permutation, used, elements, eater);
                used[n] = false;
            }
    }
}
```

Listing 4.46: Schnelles Generieren von Permutationen ohne Erzeugen temporärer Collections.

Die beiden Maßnahmen zeigen Wirkung, denn das Programm läuft ungefähr achtmal schneller als vorher:

```
$ java PermutationCounter 1 2 3 4 5 6 7 8 9 10 11
39916800
16887 millis
```

In der Literatur finden sich viele Algorithmen zum Generieren von Permutationen, die schneller arbeiten und trickreich vorgehen. Hier geht es allerdings um leicht nachvollziehbare rekursive Methoden in Java, nicht um Algorithmik.

4.5 Memoizing

4.5.1 Redundante Methodenaufrufe

Die folgende Klasse ist von `Fibonacci` (Listing 4.21) abgeleitet und redefiniert `fib`.

Wiederholte rekursive Aufrufe mit gleichem Ergebnis

Die neue Methode berechnet das Ergebnis weiterhin mit der Basisklassenmethode `super.fib`, protokolliert aber zusätzlich Aufruf und Rückkehr:

```
public class FibonacciTraced extends Fibonacci {
    private String indent = "";

    public long fib(int n) {
        System.out.printf("%sfib(%d) called%n", indent, n);
        indent += "  ";
        long result = super.fib(n);
        indent = indent.substring(2);
        System.out.printf("%sfib(%d) returns %d%n", indent, n, result);
        return result;
    }
    // main wie in der Basisklasse ...
}
```

Listing 4.47: Abgeleitete Klasse zur Berechnung von Fibonaccizahlen mit Protokoll der Methodenaufrufe.

Der Aufruf von $fib(6)$ macht sichtbar, dass die Methode `fib` immer wieder mit den gleichen Argumenten aufgerufen wird:

```
$ java FibonacciTraced 6
fib(6) called
  fib(5) called
    fib(4) called
      fib(3) called
        fib(2) called
        fib(2) returns 1
        fib(1) called
        fib(1) returns 1
      fib(3) returns 2
      fib(2) called
      fib(2) returns 1
    fib(4) returns 3
    fib(3) called
      fib(2) called
      fib(2) returns 1
      fib(1) called
      fib(1) returns 1
    fib(3) returns 2
  fib(5) returns 5
  fib(4) called
    fib(3) called
      fib(2) called
      fib(2) returns 1
      fib(1) called
      fib(1) returns 1
    fib(3) returns 2
    fib(2) called
    fib(2) returns 1
  fib(4) returns 3
```

```
fib(6) returns 8
fib(6) = 8
```

In der Ausgabe erkennt man beispielsweise zwei Aufrufe von `fib(4)`. `fib`-Aufrufe mit dem gleichen Argument liefern auch immer das gleiche Ergebnis, weil `fib` keine anderen Informationen als die Parameterwerte verwendet.

Gleiches Ergebnis mit gleichen Argumenten

Gedächtnis für Ergebnisse

Es liegt nahe, sich die Ergebnisse von `fib`-Aufrufen zu merken und bei späteren Aufrufen mit den gleichen Argumenten sofort zurückzuliefern, ohne die Berechnung zu wiederholen. Diese Erweiterung um ein „Gedächtnis“ für Ergebnisse wird als **Memoizing** bezeichnet. Die Idee ähnelt einem Cache für langsameren Massenspeicher im Betriebssystem.

Speichern von Argumenten und Ergebnissen

Die folgende Klasse `FibonacciCached` definiert eine Map, die Argumente (`Integer`) auf Ergebnisse (`Long`) abbildet. Zu Beginn ist die Map leer. Bei jedem Aufruf von `fib` wird zuerst überprüft, ob die Map bereits einen Eintrag zum aktuellen Argument enthält.

- Falls nein, wird das Ergebnis mit einem Aufruf von `super.fib` neu berechnet und dann aufgezeichnet.
- Falls ja, liefert die Methode das früher gespeicherte Ergebnis ohne neuen Aufruf der Basisklassenmethode sofort zurück.

```
import java.util.*;

public class FibonacciCached extends Fibonacci {
    private final Map<Integer, Long> cache = new HashMap<>();

    public long fib(int n) {
        Long result = cache.get(n);
        if(result == null) {
            result = super.fib(n);
            cache.put(n, result);
        }
        return result;
    }
}
```

Listing 4.48: Rekursive Berechnung der Fibonaccizahlen mit Cache für früher berechnete Werte.

Ein neuer Aufruf von `FibonacciTraced`, das jetzt nicht mehr direkt von `Fibonacci`,

Protokoll zeigt Wirkung des Speichers

sondern von `FibonacciCached` abgeleitet ist, macht die Wirkung des Memoizing sichtbar:

```
$ java FibonacciTraced 6
fib(6) called
  fib(5) called
    fib(4) called
      fib(3) called
        fib(2) called
        fib(2) returns 1
        fib(1) called
        fib(1) returns 1
      fib(3) returns 2
      fib(2) called
      fib(2) returns 1
    fib(4) returns 3
    fib(3) called
    fib(3) returns 2
  fib(5) returns 5
  fib(4) called
  fib(4) returns 3
fib(6) returns 8
fib(6) = 8
```

Performance vergleichbar mit iterativer Fassung

Wie drastisch die Maßnahme tatsächlich durchschlägt, zeigt eine entsprechend modifizierte Fassung von `FibonacciInstrumented` (Listing 4.22):

```
$ java FibonacciInstrumented 50
fib(1) = 1, millis = 33, calls = 1
fib(2) = 1, millis = 1, calls = 1
fib(3) = 2, millis = 1, calls = 3
fib(4) = 3, millis = 1, calls = 5
fib(5) = 5, millis = 1, calls = 7
...
fib(45) = 1134903170, millis = 2, calls = 87
fib(46) = 1836311903, millis = 2, calls = 89
fib(47) = 2971215073, millis = 1, calls = 91
fib(48) = 4807526976, millis = 2, calls = 93
fib(49) = 7778742049, millis = 1, calls = 95
```

Die Performance dieser Methode reicht an die der iterativen Fassung `FibonacciForward` (Listing 4.23) heran, obwohl die eigentliche Berechnung der Fibonaccizahlen weiterhin direkt der einfachen und klaren mathematischen Definition folgt.

Globaler Cache

Cache für alle Methodenaufrufe

Bisher baut jedes `FibonacciCached`-Objekt seinen eigenen Cache auf. Allerdings

berechnen alle Objekte *dieselben* Fibonaccizahlen und können sich daher einen gemeinsamen Cache teilen. Eine einfache Maßnahme ist die Definition von `cache` als Klassenvariable:

```
import java.util.*;

public class FibonacciStaticCached extends Fibonacci {
    private static final Map<Integer, Long> cache = new HashMap<>();

    public long fib(int n) {
        Long result = cache.get(n);
        if(result == null) {
            result = super.fib(n);
            cache.put(n, result);
        }
        return result;
    }
}
```

Listing 4.49: Rekursive Berechnung der Fibonaccizahlen mit globalem Cache für früher berechnete Werte.

Weitere eingesparte Methodenaufrufe

Ein Programmstart mit `FibonacciStaticCached` zeigt, dass die Anzahl der `fib`-Aufrufe noch einmal fällt:[27]

```
$ java FibonacciInstrumented 50
fib(1) = 1, millis = 13, calls = 1
fib(2) = 1, millis = 0, calls = 1
fib(3) = 2, millis = 0, calls = 3
fib(4) = 3, millis = 1, calls = 3
fib(5) = 5, millis = 0, calls = 3
...
fib(45) = 1134903170, millis = 0, calls = 3
fib(46) = 1836311903, millis = 0, calls = 3
fib(47) = 2971215073, millis = 1, calls = 3
fib(48) = 4807526976, millis = 0, calls = 3
fib(49) = 7778742049, millis = 1, calls = 3
```

Voraussetzungen für globalen Cache

Der globale statische Cache beruht darauf, dass die Methodenaufrufe aller Objekte die gleichen Werte berechnen. Für die `Fibonacci`-Klasse trifft das zwar zu, aber verallgemeinern kann man die Idee nicht unbedingt. Beispielsweise macht schon eine kleine Erweiterung der Anforderungen den Ansatz zunichte: Ändert man die ersten beiden Zahlen von 1, 1 auf 1, 3, dann erhält man mit derselben Berechnungsformel die „Lucas-Zahlen“ 1, 3, 4, 7, 11, 18 und so weiter. Auch für beliebige andere Startwerte ergibt sich jeweils eine neue Zahlenfolge. Der Konstruktor der fol-

[27] An der ohnedies minimalen Laufzeit lässt sich diese Vereinfachung nicht mehr ablesen.

genden Klasse `AnyFibonacci`[28] akzeptiert zwei Startwerte. Die Methode `fib` liefert Elemente der betreffenden Folge:[29]

```
public class AnyFibonacci {
    private final int first;

    private final int second;

    public AnyFibonacci(int first, int second) {
        this.first = first;
        this.second = second;
    }

    public long fib(int n) {
        if(n <= 1)
            return first;
        if(n == 2)
            return second;
        return fib(n - 1) + fib(n - 2);
    }

    public static void main(String... args) {
        AnyFibonacci fibonacci = new AnyFibonacci(Integer.parseInt(args[0]),
                Integer.parseInt(args[1]));
        for(int n = 1; n < Integer.parseInt(args[2]); n++)
            System.out.printf("fib(%d) = %d%n", n, fibonacci.fib(n));
    }
}
```

Listing 4.50: Berechnung von Zahlenfolgen mit der Fibonacciformel, aber unterschiedlichen Startwerten.

Das Programm kann auch die Lucas-Folge berechnen:

```
$ java AnyFibonacci 1 3 10
fib(1) = 1
fib(2) = 3
fib(3) = 4
fib(4) = 7
fib(5) = 11
fib(6) = 18
fib(7) = 29
fib(8) = 47
fib(9) = 76
```

[28] Der Namensbestandteil „Fibonacci“ signalisiert hier nur noch die Berechnungsformel und nicht mehr den Namen der generierten Zahlenfolge.

[29] Die Methode `fib` ist immer noch rein funktional, weil sie außer den Parametern nur Konstanten verwendet.

Ein statischer Cache würde bei `AnyFibonacci` zu falschen Ergebnissen führen, weil nicht alle Objekte die gleiche Folge berechnen.

Implementierung des Caches

Komplexere Cache-Struktur für Ackermann-Funktion

Memoizing beschleunigt auch die Ackermann-Implementierung `Ackermann` (Listing 4.24). Als Cache reicht eine einfache Map aber nicht mehr aus, weil Paare von Argumentwerten auf ein Ergebnis abgebildet werden müssen. Das Problem kann auf unterschiedliche Art gelöst werden. Eine Möglichkeit ist eine zweistufige Map: Die „äußere“ Map bildet das erste Argument auf eine weitere „innere“ Map ab, die das zweite Argument auf das Ergebnis abbildet. Diese Map kann so definiert werden:

```
private final Map<Long, Map<Long, Long>> cache = new HashMap<>();
```

Die Überprüfung des Caches läuft in zwei Stufen ab:

1. Das erste Argument (`m`) liefert als Schlüssel der äußeren Map (`cache`) eine innere Map oder `null`. Bei `null` muss eine neue innere Map erzeugt und in die äußere Map eingefügt werden.
2. Das zweite Argument (`n`) liefert als Schlüssel der inneren Map (`inner`) das Ergebnis oder `null`. Bei `null` wird das Ergebnis mit einem Aufruf an `super.ack` berechnet und in die innere Map eingefügt.

Die Implementierung der Methode sieht folgendermaßen aus:

```
import java.util.*;

public class AckermannCached extends Ackermann {
    private final Map<Long, Map<Long, Long>> cache = new HashMap<>();

    public long ack(long m, long n) {
        Map<Long, Long> inner = cache.get(m);
        if(inner == null) {
            inner = new HashMap<>();
            cache.put(m, inner);
        }
        Long result = inner.get(n);
        if(result == null) {
            result = super.ack(m, n);
            inner.put(n, result);
        }
        return result;
    }
}
```

Listing 4.51: Berechnung der Ackermann-Funktion mit Cache.

Auswirkung des Caches

Der Cache wirkt sich deutlich aus, wie ein Aufruf der instrumentierten Klasse zeigt:

```
$ java -Xss4M AckermannInstrumented 4 1
ack(4, 1) = 65533, millis = 781, calls = 196625, max depth = 16389
```

Die Laufzeit fällt von knapp zwei Minuten auf weniger als eine Sekunde. Zwar sind immer noch viele Tausend Methodenaufrufe zur Berechnung von $ack(4,1)$ nötig, aber das ist nur ein Bruchteil der ursprünglichen Fassung. An der Überschreitung des Wertebereichs bei moderaten Argumenten ändert der Cache natürlich nichts.

Cache-Struktur zum Memoizing der Takeuchi-Funktion

Am Beispiel von `Takeuchi` (Listing 4.25) soll ein anderer Ansatz zur Implementierung des Caches gezeigt werden. Ein Caching entsprechend zur Ackermann-Funktion macht hier eine dreistufige Map nötig, deren Verwaltung einigen Code erfordert. Eine einfache Map reicht dagegen aus, wenn eine Kombination von drei Parameterwerten als Schlüssel verwendet werden kann. Die folgende Klasse `LongTupel` packt `long`-Werte in ein unveränderliches Objekt.[30] `equals` und `hashCode` werden an die Klasse `Arrays` delegiert.

```
import java.util.*;

public class LongTupel {
    private final long[] values;

    public LongTupel(long... values) {
        this.values = values;
    }

    @Override public boolean equals(Object any) {
        if(any == null || any.getClass() != LongTupel.class)
            return false;
        LongTupel that = (LongTupel)any;
        return Arrays.equals(values, that.values);
    }

    @Override public int hashCode() {
        return Arrays.hashCode(values);
    }

}
```

Listing 4.52: Tupel mehrerer `long`-Werte.

Die Memoizing-Implementierung `TakeuchiCached` verwendet die Klasse `LongTupel` als Schlüssel für eine einfache `HashMap`:

[30] Genau genommen könnte der Inhalt eines `LongTupel` von außen manipuliert werden, wenn der Konstruktor ein fertiges Array als Argument erhält und der Aufrufer dieses Array nachträglich modifiziert. Diese Lücke wird hier hingenommen.

```
import java.util.*;

public class TakeuchiCached extends Takeuchi {
    private final Map<LongTupel, Long> cache = new HashMap<>();

    public long tak(long x, long y, long z) {
        LongTupel at = new LongTupel(x, y, z);
        Long result = cache.get(at);
        if(result == null) {
            result = super.tak(x, y, z);
            cache.put(at, result);
        }
        return result;
    }
}
```

Listing 4.53: Berechnung der Takeuchi-Funktion mit Cache auf der Basis von `long`-Tupeln.

Jetzt lässt sich ein Wert der Takeuchi-Funktion bestimmen, der ohne Memoizing eine Rechenzeit von mehreren Stunden erfordert:

```
$ java TakeuchiInstrumented 32 24 6
tak(32, 24, 6) = 7, millis = 16, calls = 4569, max depth = 44
```

4.5.2 Bedingungen für Memoizing

Anwendungsmöglichkeiten von Memoizing

Memoizing ist unter zwei Voraussetzungen möglich:

- Das Ergebnis der betreffenden Methode hängt nur von den Argumenten und von sonst keinen veränderlichen Daten ab. Konstanten sind kein Problem.
- Die Methode löst keine Seiteneffekte aus, das heißt, ein Aufruf hat keine von außen erkennbare Wirkung.

Rein funktionale Methoden

Eine solche Methode wird als „rein funktional“ (*pure functional*) bezeichnet. Eine rein funktionale Methode liefert beim Aufruf mit den gleichen Argumenten immer das gleiche Ergebnis. Abgesehen vom Verbrauch an Rechenleistung ist nicht nachweisbar, ob und gegebenenfalls wie oft sie aufgerufen wurde.[31]

Als einzige Möglichkeit sich mitzuteilen bleibt einer rein funktionalen Methode die Rückgabe eines Ergebnisses.[32] Aber auch bei rein funktionalen Methoden ist Memoizing nicht immer sinnvoll. Der Cache wirkt sich erst bei mehreren Aufrufen mit

[31] Programmiersprachen, die ausschließlich rein funktional arbeiten, werden als „rein funktionale Programmiersprachen“ bezeichnet. Java und die meisten anderen populären Sprachen zählen nicht dazu.

[32] Ein rein funktionale `void`-Methode wäre sinnlos, weil sie nichts bewirken und nichts liefern könnte.

den gleichen Argumenten aus. Eine Methode, die mit immer wieder neuen Argumenten aufgerufen wird, baut nur einen Cache auf, ohne ihn zu nutzen. Sie läuft dadurch sogar *langsamer* als eine einfachere Implementierung ohne Memoizing.

Parameterliste entscheidend für Memoizing

Die Chancen für wiederkehrende Argumentwerte hängen von der Anzahl der Parameter und deren Typen ab. Die besten Aussichten haben Methoden mit wenigen Parametern, wenn diese auch noch kleine Wertebereiche haben, wie zum Beispiel `boolean`, `byte` und `char`. Bei Floatingpoint-Typen können die unvermeidlichen Rundungsfehler die Wirkung von Memoizing zunichte machen. Bei Strings und vielen anderen Referenztypen gibt es meistens so viele verschiedene Argumentwerte, dass Memoizing unrentabel wird.

Auf der anderen Seite kommt die Natur der Rekursion der Wirksamkeit von Memoizing entgegen: Die Argumente konvergieren in der Regel zur Abbruchbedingung hin.[33] Dadurch steigt die Wahrscheinlichkeit, dass zumindest in der „Nähe“ der Abbruchbedingung wiederholte Aufrufe mit gleichen Argumenten stattfinden.

4.5.3 Implementierung von Caches

Platzbedarf für Caches

Memoizing beschleunigt rekursive Methoden unter bestimmten Voraussetzungen nahezu auf die Geschwindigkeit iterativer Fassungen. Die dazu nötigen Caches für Ergebnisse kosten allerdings Platz und konkurrieren mit den „echten“ Daten[34] des Programms. Das Volumen der Caches ist zudem schwer kalkulierbar und hängt vom einzelnen Programmverlauf ab. Wenn es eng wird, sollte ein Programm Caches aufgeben, um Platz zu schaffen. Die Voraussetzung dafür erschließt ein Blick auf die Speicherverwaltung von Java.

Stack und Heap

Speicherbereiche eines Java-Programms

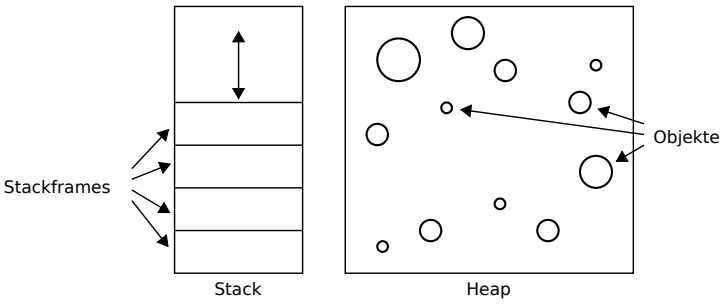
Der größte Speicherbereich eines laufenden Java-Programms ist der „Heap“, der Objekte aufnimmt. Jedes `new` in einem Java-Programm verbraucht für das neu geschaffene Objekt etwas Speicherplatz auf dem Heap.[35] Der zweite, meist kleine-

[33] Funktionen, von denen man nicht weiß, ob sie immer auf die Abbruchbedingung zusteuern, heißen „offen rekursiv“. Eine rekursive Implementierung der „Collatz-Folge“ (Einzelheiten zum Beispiel auf `http://de.wikipedia.org/wiki/Collatz-Problem`) ist offen rekursiv.

[34] „Echt“ bedeutet hier, dass diese Daten nicht neu berechnet werden können. Im Gegensatz dazu könnten die „redundanten“ Daten in den Caches rekonstruiert werden, wenn auch möglicherweise mit einigem Aufwand.

[35] Die meisten Objekte werden durch ein explizites `new` geschaffen, das zu einem Konstruktoraufruf führt. Ein implizites `new` steckt in String- und Array-Literalen sowie im Boxing primitiver Werte in Wrapper-Objekten. Neue Objekte auf dem Heap können aber auch ohne Konstruktoraufrufe entstehen, wie zum Beispiel mit `clone` oder bei der Deserialisierung, wie in `CounterClones` (Listing 2.8).

re Speicherbereich ist der „Stack“, auf dem für die Dauer von Methodenaufrufen Stackframes gestapelt werden. Dazu kommt noch der unveränderliche Code sowie konstante Datenstrukturen, die im Moment keine Rolle spielen.



Garbage-Collector

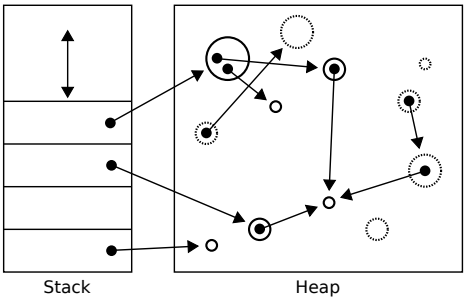
Regelmäßige Bereinigung des Heaps

Alle realistischen Java-Programme erzeugen Objekte und verbrauchen damit ständig Platz im Heap, der über kurz oder lang aufgebraucht sein wird. Dann startet automatisch der Garbage-Collector (GC) und versucht, wieder Platz zu gewinnen.[36] Er überprüft alle Objekte auf dem Heap und stellt fest, ob das laufende Programme sie noch jemals verwenden könnte. Dazu beginnt der Garbage-Collector mit dem Stack, auf dem die lokalen Variablen und Argumente aller momentan aufgerufenen Methoden in Stackframes gestapelt sind. Alle vom Stack aus referenzierten Objekte könnte das Programm noch ansprechen.[37] Das betrifft auch alle mittelbar erreichbaren Objekte, also beispielsweise Array- und Collection-Elemente, Objekt- und Klassenvariablen in referenzierten Objekten. Die Menge *aller* so erreichbaren Objekte bildet das **Working-Set** des Programms im gegebenen Moment. In der folgenden Skizze sind Objekte im Working-Set durchgezogen umrandet, Objekte außerhalb des Working-Set gestrichelt:

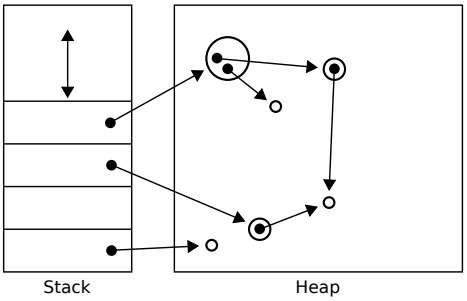
Working-Set = Menge aller erreichbaren Objekte

[36] Die Arbeitsweise des Garbage-Collectors ist hier vereinfacht dargestellt. In Wahrheit geht er trickreicher zu Werke. Das konkrete Verfahren wurde in den verschiedenen Java-Versionen schon mehrfach umgestellt.

[37] Primitive Werte spielen keine Rolle.



Die Objekte des Working-Set muss der Garbage-Collector verschonen, aber alle anderen kann er freigeben und damit wieder Platz im Heap zurückgewinnen:



Demonstration des Garbage-Collectors

Das folgende Programm definiert zunächst einen String, der nur ein Zeichen lang ist. In einer Endlosschleife konkateniert es den String mit sich selbst und erzeugt so immer neue Strings mit der jeweils doppelten Länge:

```
public class ObjectsForever {
    public static void main(String... args) {
        String string = "x";
        while(true)
            string += string;
    }
}
```

Listing 4.54: Allozieren immer längerer Strings, bis der Heap aufgebraucht ist.

Die Variable `s` referenziert immer den letzten, längsten String. Das Programm muss über kurz oder lang unweigerlich mit einem `OutOfMemoryError` abstürzen:

```
$ java ObjectsForever
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
...
```

Zu jedem Zeitpunkt ist aber nur jeweils der jüngste, längste String im Working-Set, weil die lokale Variable `s` Bestandteil des (einzigen) Stackframes des `main`-Aufrufs ist. Die vorher verwendeten Strings sind dagegen nicht mehr erreichbar und aus dem Working-Set herausgefallen.

Sichtbare Aufrufe des Garbage-Collectors

Normalerweise arbeitet der Garbage-Collector unsichtbar im Hintergrund und macht sich allenfalls durch kurze Pausen im laufenden Programm bemerkbar.[38] Der Schalter `-verbose:gc` beim Start der JVM macht GC-Aufrufe auf dem Bildschirm sichtbar. Am Absturz ändert das nichts. Jeder GC-Lauf gibt alle Strings bis auf den letzten frei, bis die Stringlänge schließlich die Größe des Heaps überschreitet.

```
$ java -verbose:gc ObjectsForever
[GC 12992K->4491K(61120K), 0.0080330 secs]
[GC 648375K->633112K(714560K), 0.0130240 secs]
[GC 633112K->633112K(714560K), 0.0149860 secs]
...
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

Explizite Garbage-Collection

Aufruf des Garbage-Collectors durch das Programm

Der Garbage-Collector kann mit der folgenden Anweisung jederzeit explizit gestartet werden. Wenn die Methode `gc` zurückkehrt, hat der GC alles getan, um Platz zu schaffen.

```
System.gc();
```

Die Methode dient in erster Linie zu Testzwecken. Normaler Code ruft sie kaum auf. Allenfalls kann ein Programm versuchen, vor einem zeitkritischen Abschnitt die Gefahr einer GC-Unterbrechung zu verringern. Ausschließen lässt sie sich allerdings nicht.

Referenzarten

Arten von Referenzen

Java kennt vier verschiedene Arten von Referenzen,[39] die der Garbage-Collector unterschiedlich behandelt.

[38] Manche Anwendungen können auch diese kurzen Pausen nicht tolerieren und brauchen besondere Garbage-Collectoren, die ständig „mitlaufen“. Das bremst zwar die Gesamtgeschwindigkeit des Programms ab, vermeidet aber Aussetzer.

[39] Hier ist die Rede von „*Arten* von Referenzen“ und von „Referenz*typen*“. Letzterer bezeichnet Interfaces und Klassen im Gegensatz zu primitiven Typen.

Strong-Reference

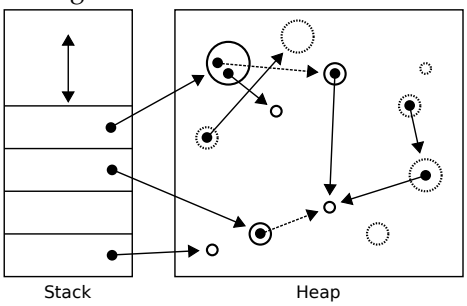
Normale Objekte

Strong-References sind ganz normale Referenzen auf Objekte. Der Garbage-Collector lässt alle Objekte am Leben, die über eine durchgehende Kette von Strong-References erreichbar sind.

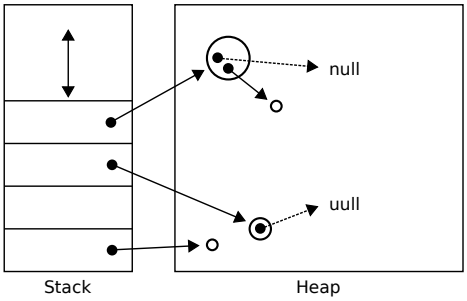
Soft-Reference

Verzichtbare Objekte mit möglichst langer Lebensdauer

Auch Soft-References referenzieren Objekte. Der Garbage-Collector *kann* Objekte, die nur über Soft-References erreichbar sind, freigeben und Referenzen durch `null` ersetzen.
In der folgenden Skizze sind zwei Soft-References durch gestrichelte Pfeile dargestellt:



Der GC löscht alle Objekte, die entweder überhaupt nicht oder nur über Soft-References erreichbar sind. Er ersetzt die Referenzziele von Soft-References durch `null`:



In der Praxis wartet der GC bis zum letzten Moment und räumt „soft-referenzierte“ Objekte erst dann ab, wenn der Heap aufgebraucht ist. Ein expliziter Aufruf von `System.gc()` „auf Verdacht“ lässt Soft-References dagegen in der Regel unberührt.

Weak-Reference

Verzichtbare Objekte mit kurzfristiger Lebensdauer

Weak-References funktionieren wie Soft-References. Allerdings *muss* der Garbage-Collector Objekte, die nur noch über Weak-References erreichbar sind, freigeben und Referenzen durch `null` ersetzen. Dabei spielt es keine Rolle, wie ausgelastet der Heap ist.

Phantom-Reference

Bereits entfernte Objekte

Phantom-References haben nichts mit Soft- und Weak-References zu tun. Sie referenzieren Objekte, die der Garbage-Collector bereits freigegeben hat, und haben immer den Wert `null`. Allerdings existieren sie noch als Referenz und können zum Beispiel abgezählt werden. Phantom-References werden hier nicht betrachtet.

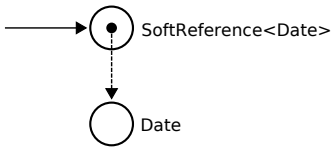
Strong-References sind in Java allgegenwärtig und brauchen weiter keine Unterstützung.

Konstruktor für `SoftReference`

Eine **Soft-Reference** erzeugt der Konstruktor der Klasse `SoftReference` im Package `java.lang.ref`. Einen anderen Weg gibt es nicht. Die folgende Anweisung ruft den `SoftReference`-Konstruktor auf und übergibt ihm ein `Date`-Objekt. Das zurückgelieferte `SoftReference`-Objekt enthält eine Strong-Reference auf das Argument.

```
SoftReference<Date> sr = new SoftReference<>(new Date());
```

Diese Skizze zeigt die Konstruktion. Der gestrichelte Pfeil markiert die Soft-Reference:



Die Methode `get` holt das referenzierte Objekt aus einer Soft-Reference heraus:

```
System.out.println(sr.get());
```

Nachweis der Eigenschaft von Soft-References

Das folgende Programm erzeugt eine Soft-Reference auf ein `Date`-Objekt, auf das *sonst keine* Strong-Reference verweist. Nach einer Kontrollausgabe wird das Programm `ObjectsForever` (Listing 4.54) aufgerufen, das den ganzen Heap aufbraucht. Das abschließende `finally` gibt die Referenz noch einmal aus.

```
import java.lang.ref.*;
import java.util.*;

public class SoftReferenceOutOfMemory {
    public static void main(String... args) {
        SoftReference<Date> softReference = new SoftReference<>(new Date());
        System.out.println(softReference.get());
        try {
            ObjectsForever.main(args);
```

```
        }
        finally {
            System.out.println(softReference.get());
        }
    }
}
```

Listing 4.55: Freigabe von Soft-References unter Speichermangel.

Nach dem `OutOfMemoryError` enthält die Soft-Reference nur noch `null`:

```
$ java SoftReferenceOutOfMemory
Sun Oct 30 06:48:42 CET 2011
null
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

Das nächste Programm provoziert keinen Speichermangel, sondern ruft den Garbage-Collector mutwillig auf:

```
import java.lang.ref.*;
import java.util.*;

public class SoftReferenceGC {
    public static void main(String... args) {
        SoftReference<Date> softReference = new SoftReference<>(new Date());
        System.out.println(softReference.get());
        System.gc();
        System.out.println(softReference.get());
    }
}
```

Listing 4.56: Keine Freigabe von Soft-References durch den Garbage-Collector ohne Speichermangel.

Diesmal verschont der GC die Soft-Reference, weil keine Speicherknappheit herrscht:

```
$ java SoftReferenceGC
Sun Oct 30 06:48:57 CET 2011
Sun Oct 30 06:48:57 CET 2011
```

Arbeitsweise von `WeakReference`

Die Klasse `WeakReference` funktioniert wie die Klasse `SoftReference`, installiert aber eine **Weak-Reference**. Das Testprogramm `WeakReferenceOutOfMemory` verhält sich genauso wie `SoftReferenceOutOfMemory`. Anders als eine Soft-Reference löscht der GC aber eine Weak-Reference *in jedem Fall*, wie die Ausgabe des Programms `WeakReferenceGC` zeigt:

```
$ java WeakReferenceGC
Sun Oct 30 06:49:11 CET 2011
null
```

`WeakHashMap` als Cache

`WeakHashMap` mit Weak-References

Das Package `java.util` stellt Collection-Klassen zur Verfügung, darunter auch `WeakHashMap`.[40] Eine `WeakHashMap` funktioniert wie eine `HashMap`, kapselt aber alle Schlüssel in Weak-References.

Im Unterschied zu einer normalen `HashMap` löscht eine Garbage-Collection alle Einträge, wenn die betreffende Weak-Reference verschwindet. Diese Klasse eignet sich als Cache, beispielsweise für Memoizing. Weil `WeakHashMap` ebenso wie `HashMap` das Interface `Map` implementiert, ist die Umstellung zunächst trivial.

Dennoch verhalten sich `WeakHashMaps` etwas anders als die übrigen Maps:

- Nur die Schlüssel, nicht die Werte, sind in Weak-References gekapselt. Strong-References auf den Schlüssel halten einen Eintrag am Leben, Strong-References auf den Wert nicht.

  Weak-References halten Schlüssel, keine Werte
- Eine `WeakHashMap` kollabiert bei *jedem* `gc`-Aufruf, ob automatisch unter Speichermangel oder explizit ausgelöst. Unter Umständen ist das so nicht gewünscht.[41]
- Manche Objekte leben außerhalb des Heaps und damit auch außerhalb der Reichweite des Garbage-Collectors. Dazu zählen beispielsweise String-Literale, die der Compiler als Konstanten in den Bytecode einbaut. Auch Wrapper-Objekte mit kleinen ganzzahligen Werten[42] und `Boolean`-Objekte[43] werden im Laufzeitsystem getrennt vom Heap verwaltet. Solche Objekte sind als Schlüssel einer `WeakHashMap` nutzlos, weil die Einträge nie freigegeben werden.

  Schlüssel mit ewigem Leben

  Das folgende Beispielprogramm illustriert dieses Verhalten:

```
import java.util.*;

public class FunnyWeakMap {
    public static void main(String... args) {
        WeakHashMap<Object, Integer> map = new WeakHashMap<>();
        map.put("foo", 1);
        map.put(new String("bar"), 2);
```

[40] Es gibt zwei weitere ähnliche Klassen, `sun.misc.SoftCache` und `com.sun.beans.WeakCache`, deren langfristige Verfügbarkeit aber ungewiss ist und die deshalb hier nicht diskutiert werden.

[41] Es gibt im Package `java.util` keine `SoftHashMap`.

[42] Die Bedeutung von „klein“ ist nicht festgelegt. Derzeit gelten Werte von -128 bis 127 als „klein“. In künftigen Java-Versionen kann sich der Bereich ändern.

[43] Davon gibt es ohnedies nur zwei.

```
        map.put(127, 3);
        map.put(128, 4);

        System.out.println(map);
        System.gc();
        System.out.println(map);
    }
}
```

Listing 4.57: Freigabe von Weak-References durch den GC, keine Freigabe aus statischen Objekten.

Die Schlüssel `"foo"` und `127` leben nicht auf dem Heap und entgehen der Garbage-Collection:

```
$ java FunnyWeakMap
{bar=2, foo=1, 128=4, 127=3}
{foo=1, 127=3}
```

Auch einfache Methoden möglicherweise langsam

- Eine `WeakHashMap` kann mitten im Programmablauf kollabieren, obwohl sie der Anwender nur liest oder überhaupt nicht berührt. Das hat unter anderem zur Folge, dass Aufrufe trivialer Methode wie `size` und `isEmpty` einigen Aufwand verursachen können und möglicherweise unerwartet langsam arbeiten.

Iteratoren in einer `WeakHashMap`

- Iteratoren auf Views einer `WeakHashMap` überstehen Änderungen, die der Garbage-Collector auslöst. Sie werfen insbesondere keine `ConcurrentModificationException`. Das folgende Beispielprogramm druckt die Map-Einträge mit einer *foreach*-Schleife aus und verwendet dabei einen impliziten Iterator. Beim Aufruf von `gc` *im Rumpf* der Schleife schrumpft die Map, während die Schleife läuft:

```
import java.util.*;

public class FunnyWeakMapLoop {
    public static void main(String... args) {
        WeakHashMap<Object, Integer> map = new WeakHashMap<>();
        map.put(new String("bar"), 1);
        map.put(128, 2);
        map.put("retained", 3);

        for(Map.Entry<Object, Integer> entry: map.entrySet()) {
            System.out.printf("%s -> %s%n", entry.getKey(), entry.getValue());
            System.gc();
        }
    }
}
```

Listing 4.58: Iteration über eine `WeakHashMap`.

Der erste GC löscht die ersten zwei Einträge aus der Map. Der erste Eintrag ist zu diesem Zeitpunkt bereits ausgegeben, aber der zweite verschwindet ohne den Iterator zu zerstören. Die Schleife läuft daher fehlerfrei zu Ende.

```
$ java FunnyWeakMapLoop
bar -> 1
retained -> 3
```

Ein `hasNext`-Aufruf eines Iterators, der `true` liefert, speichert eine Strong-Reference auf den Wert im Iterator. Ein nachfolgendes `next` wirft keinesfalls eine `NoSuchElementException`, auch wenn dazwischen der Garbage-Collector aktiv wurde.

Zusammenfassung

- Rekursive Methoden enthalten einen **Selbstaufruf** und ein **Abbruchkriterium**.
- Der **Laufzeitstack** begrenzt die maximale **Rekursionstiefe**.
- Rekursion und **Iteration** (Schleifen) sind **äquivalent**.
- Eine **endrekursive Methode** kehrt mit dem Ergebnis des rekursiven Aufrufs zurück. Endrekursion kann mechanisch in Iteration transformiert werden.
- **Lineare**, **verzweigte** und **verschachtelte** Rekursion haben charakteristische Eigenschaften.
- **Memoizing** kann rekursive Methoden durch Aufzeichnen und Wiederverwenden von Ergebnissen beschleunigen, wenn sie nur von Argumenten und Konstanten abhängen.
- Die JVM löscht `SoftReference`- und `WeakReference`-Objekte bei Platzmangel. Sie eignen sich für Caches und andere temporäre Datenstrukturen.

Aufgaben

Aufgabe 1: Zahlen-Eigenschaften

Quersumme

Das folgende Programm berechnet die **Quersumme** der Zahl, die auf der Kommandozeile angegeben ist:

```
public class ChecksumLoop {
    public static void main(String... args) {
        int n = Integer.parseInt(args[0]);
        int sum = 0;
        while(n > 0) {
            int digit = n%10;
            sum += digit;
```

```
            n /= 10;
        }
        System.out.println(sum);
    }
}
```

Listing 4.59: Berechnet iterativ die Quersumme.

Der folgende Aufruf zeigt ein Beispiel:

```
$ java ChecksumLoop 238059
27
```

Schreiben Sie eine endrekursive Fassung dieses Programms, die keine Schleife enthält, in deren Quelltext also weder `for` noch `while` vorkommt.

Primzahlen

Das folgende Programm sucht **Primzahlen** in einem Zahlenbereich und gibt sie aus. Der Test von potenziellen Teilern einer Zahl kann bei der Wurzel abgebrochen werden, weil zu jedem Teiler über der Wurzel auch einer unter der Wurzel existieren muss.

```
public class PrimeLoop {
    public static void main(String... args) {
        int upto = Integer.parseInt(args[0]);
        for(int n = 2; n < upto; n++) {
            boolean isPrime = true;
            int factor = 2;
            while(isPrime && factor*factor <= n)
                isPrime = n%factor++ > 0;
            if(isPrime)
                System.out.println(n);
        }
    }
}
```

Listing 4.60: Sucht nach Primzahlen und gibt sie aus.

Beim Aufruf mit der Obergrenze 10 erhält man die korrekten Primzahlen:

```
$ java PrimeLoop 10
2
3
5
7
```

Schreiben Sie eine endrekursive Fassung dieses Programms, die sich gleich verhält. Die rekursive Lösung funktioniert nicht mit großen Zahlen, weil dazu der Stack nicht ausreicht. Eine gute Fingerübung zur rekursiven Programmierung ist das Programm aber dennoch.

Perfekte Zahlen

Eine Zahl ist „perfekt“, wenn die Summe ihrer Teiler (einschließlich der 1, ohne sie selbst) genauso groß ist wie sie selbst. Zum Beispiel ist `6` perfekt, denn $1 + 2 + 3 = 6$. Das folgende Programm sucht **perfekte Zahlen** bis zu einer festgelegten Obergrenze.

```
public class PerfectLoop {
    public static void main(String... args) {
        int n = 2;
        int upto = Integer.parseInt(args[0]);
        while(n < upto) {
            int factor = 2;             // potenzieller Teiler
            int sum = 1;                // Teilersumme
            while(factor*factor < n) { // Teiler bis zur Wurzel ausprobieren
                if(n%factor == 0) {   // Teiler gefunden
                    sum += factor;      // Teiler zur Summe addieren
                    sum += n/factor;  // weiteren Teiler addieren
                }
                factor++;               // nächster potenzieller Teiler
            }
            if(factor*factor == n)    // Quadratzahl?
                sum += factor;          // Wurzel addieren
            if(n == sum)                // perfekte Zahl?
                System.out.println(n);  // ja, ausgeben
            n++;
        }
    }
}
```

Listing 4.61: Iterative Suche nach perfekten Zahlen.

Ein Aufruf mit der Obergrenze 10 findet die 6:

```
$ java PerfectLoop 10
6
```

Schreiben Sie eine endrekursive Fassung dieses Programms, die die ersten vier perfekten Zahlen ausgibt. Vergrößern Sie dazu den Laufzeitstack mit dem Schalter `-Xss1M` auf 1 Megabyte.

Fröhliche Zahlen

Um festzustellen, ob eine Zahl n „fröhlich“ ist, addiert man die Quadrate ihrer Ziffern. Mit der Summe verfährt man wieder genauso und so weiter. Trifft man dabei irgendwann auf die 1, dann war die ursprüngliche Startzahl „fröhlich“. Endet man dagegen bei einer 4, so war sie „traurig“.[44] Beispielsweise ist 7 eine fröhliche Zahl, denn

$$7^2 = 49$$
$$4^2 + 9^2 = 16 + 81 = 97$$
$$9^2 + 7^2 = 81 + 49 = 130$$
$$1^2 + 3^2 + 0^2 = 1 + 9 + 0 = 10$$
$$1^2 + 0^2 = 1$$

Das folgende Programm sucht bis zu einer gegebenen Obergrenze nach **fröhlichen Zahlen**.

```
public class HappyLoop {
    public static void main(String... args) {
        final int upto = Integer.parseInt(args[0]);
        int n = 1;
        while(n <= upto) {
            int next = n;
            while(next != 1 && next != 4) {
                int sum = 0;
                while(next > 0) {
                    final int d = next%10;
                    sum += d*d;
                    next /= 10;
                }
                next = sum;
            }
            if(next == 1)
                System.out.println(n);
            n++;
        }
    }
}
```

Listing 4.62: Programm zur Suche nach fröhlichen Zahlen.

Schreiben Sie eine endrekursive Fassung dieses Programms.

[44] Dieser Algorithmus endet immer mit 1 oder 4. Eine Begründung findet sich auf `http://en.wikipedia.org/wiki/Happy_number`.

Aufgabe 2: Hofstadter-Zahlenfolge

Eine etwas sonderbare Zahlenfolge[45], die den Fibonaccizahlen ähnelt, ist folgendermaßen definiert:

$$Q(1) = Q(2) = 1$$
$$Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)) \text{ für } n > 2$$

Die ersten beiden Elemente sind 1. Weitere Elemente ergeben sich aus der Summe vorhergehender Elemente. Die beiden *direkt* vorhergehenden Elemente legen dabei aber nur fest, *wie weit zurück* in der Folge die tatsächlichen Summanden zu finden sind. Die Folge beginnt mit:

1, 1, 2, 3, 3, 4, 5, **5, 6**

Das nächste Element ist die Summe der Elemente, die vom Ende aus gesehen 6 beziehungsweise 5 Positionen weiter vorne stehen. Diese haben beide den Wert 3, also ist das nächste Element der Folge $3 + 3 = 6$. Das übernächste Element hat auch wieder den Wert 6, dann folgt 8 und so weiter.

Definieren Sie eine Klasse `Hofstadter` mit der Methode

```
int q(int n)
```

die das Element $Q(n)$ berechnet. Setzen Sie die rekursive Definition direkt um. Die folgende Anwendung gibt die Folge aus:

```
public class HofstadterMain {
    public static void main(String... args) {
        Hofstadter h = new Hofstadter();
        int upto = Integer.parseInt(args[0]);
        for(int n = 1; n <= upto; n++)
            System.out.printf("q(%d) = %d%n", n, h.q(n));
    }
}
```

Listing 4.63: Ausgabe der chaotischen Zahlenfolge aus Hofstadter: Gödel, Escher, Bach, S 149.

Leiten Sie von `Hofstadter` eine neue Klasse `HofstadterCache` ab, die einen Cache implementiert. Wiederholen Sie die Definition dabei nicht, sondern nutzen Sie die Basisklasse. Finden Sie das zehnmillionste Element der Folge.[46]

[45] Hofstadter: „Gödel, Escher, Bach“, Klett-Cotta, Stuttgart 1985, Seite 149.

[46] Zur Kontrolle: Es hat den Wert 5124632.

Aufgabe 3: Beamtenhierarchie

Die **Staatsregierung** besteht im Wesentlichen aus **Beamten**, von denen jeder über eine Anzahl untergebener Beamter verfügt (eventuell auch gar keine). Die Untergebenen können selbst wieder Untergebene haben und so weiter. Um zu einer eigenen Entscheidung zu kommen, braucht ein Beamter eine bestimmte Zeit. Definieren Sie eine geeignete Klasse `Beamter` mit einem Konstruktor, der die Denkzeit und die Untergebenen als Argumente erhält.

Die **Bedeutung** eines Beamten entspricht der Anzahl seiner direkten und indirekten Untergebenen. Ein Beamter ohne Untergebene, wie beispielsweise ein Professor, hat die Bedeutung 0, ist also bedeutungslos. Definieren Sie in der Klasse `Beamter` die Methode

```
int bedeutung()
```

Wenn ein Beamter um eine **Entscheidung** gebeten wird, denkt er zum einen selbstständig für die im Konstruktor festgesetzte Dauer darüber nach. Zum anderen verlangt er von seinen Untergebenen *nacheinander* ebenfalls eine Entscheidung. Wer länger braucht, der Beamte selbst oder alle Untergebenen nacheinander, bestimmt die tatsächliche Dauer der Entscheidung. Definieren Sie die Methode

```
int entscheidung()
```

die die Dauer der Entscheidung eines Beamten liefert.

Ein Beamter kann direkte oder indirekte **Untergebene** haben. Definieren Sie die Methode

```
boolean untergeben(Beamter b)
```

für die Auskunft, ob `b` ein Untergebener dieses Beamten (`this`) ist oder nicht. Zur Vereinfachung können Beamte mit dem Gleichheitsoperator verglichen werden.

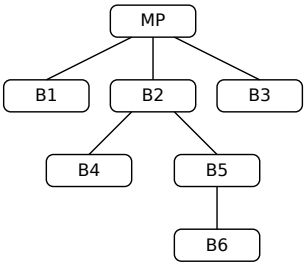
Alle Beamten unterstehen direkt oder indirekt dem **Ministerpräsidenten**, der als Konstante `MP` in der folgenden Klasse `Staatsregierung` definiert ist, ebenso wie die anderen Beamten:

```
import static java.lang.System.*;

public class Staatsregierung {
    private static final Beamter b6 = new Beamter(6);
    private static final Beamter b5 = new Beamter(5, b6);
```

```
    private static final Beamter b4 = new Beamter(4);
    private static final Beamter b3 = new Beamter(3);
    private static final Beamter b2 = new Beamter(2, b4, b5);
    private static final Beamter b1 = new Beamter(1);
    public static final Beamter MP = new Beamter(0, b1, b2, b3);

}
```

Listing 4.64: Klasse mit einigen Beamten und dem Ministerpräsidenten.

Die folgende Skizze zeigt die Struktur der Staatsregierung:



Die **Ebene** eines Beamten ergibt sich aus der Anzahl seiner Vorgesetzten, bis hinauf zum Ministerpräsidenten. Der Ministerpräsident selbst ist Beamter auf Ebene 0 (der höchsten Ebene), seine direkten Untergebenen sind Beamte auf Ebene 1 und so weiter. Definieren Sie die Methode

```
int ebene()
```

die die Ebene eines Beamten berechnet.

Die folgende `main`-Methode in der Klasse `Staatsregierung` gibt Informationen über alle Beamten aus:

```
public static void main(String[] args) {
    Beamter[] regierung = new Beamter[] {
        MP, b1, b2, b3, b4, b5, b6
    };
    for(Beamter beamter: regierung) {
        out.println(beamter);
        out.printf("\tBedeutung: %d%n", beamter.bedeutung());
        out.printf("\tEntscheidungsdauer: %d%n", beamter.entscheidung());
        for(Beamter x: regierung)
            if(beamter.untergeben(x))
                out.printf("\tVorgesetzter von: %s%n", x);
        out.printf("\tEbene: %d%n", beamter.ebene());
    }
}
```

Testen Sie Ihre Klasse `Beamter` mit dieser Anwendung.

Aufgabe 4: Tiefe Map

Eine `Map` bildet Schlüssel auf Werte ab, die selbst wieder Maps sein können. Eine solche Map könnte man als „zweidimensionale“ Map betrachten, weil sie *zwei* Schlüssel auf einen Wert abbildet. Diese Idee lässt sich auf n-stufige Maps ausdehnen.

Das folgende Interface beschreibt eine `DeepMap`, die eine *beliebige* positive Anzahl Schlüssel vom Typ `T` auf einen Wert vom Typ `U` abbildet.[47]

```
public interface DeepMap<T, U> {
    void put(U value, T... keys);
    U get(T... keys);
    int size();
    DeepMap<T, U> sub(T... keys);
    Class<?> at(T... keys);
}
```

Listing 4.65: Interface für tiefe Maps.

Die Methoden haben die folgenden Aufgaben:[48]

`void put(U value, T... keys)`
: Fügt `value` unter der Folge der Schlüssel `keys` ein. Ersetzt alle Werte unter Schlüsselfolgen, die mit `keys` beginnen.[49]

`U get(T... keys)`
: Liefert den Wert unter der Schlüsselfolge `keys` oder `null`, wenn dort kein Wert gespeichert ist.

`int size()`
: Liefert die Anzahl der Werte in der `DeepMap`.

[47] Die Werte in einer `DeepMap` dürfen keine anderen `DeepMaps` sein. Diese Konstruktion würde zu Mehrdeutigkeiten führen und soll hier ignoriert werden.

[48] `DeepMap` ist *nicht* zum Interface `java.util.Map` kompatibel, auch wenn die Methoden ähnlich benannt sind.

[49] Die Reihenfolge von Schlüssel und Wert ist in der Parameterliste dieser `put`-Methode gegenüber den entsprechenden Methoden der Maps in der Laufzeitbibliothek genau vertauscht. Diese Änderung ist nötig, weil ein Vararg-Parameter immer hinten in der Parameterliste stehen muss.

`DeepMap<T, U> sub(T... keys)`
: Liefert eine Sicht auf alle Werte, deren Schlüsselfolgen mit `keys` beginnen. In der zurückgelieferten `DeepMap` gelten nur noch die Restfolgen ohne `keys`. Liefert `null`, wenn es keine solchen Werte gibt.

`Class<?> at(T... keys)`
: Liefert das Typobjekt `U`, wenn unter der Schlüsselfolge `keys` ein Wert gespeichert ist. Liefert `DeepMap.class` oder ein abgeleitetes Typobjekt, wenn es Werte gibt, deren Schlüsselfolgen mit `keys` beginnen. Liefert `null`, wenn unter `keys` nichts gespeichert ist.

Die folgende Anwendung arbeitet mit einer `DeepMap`:

```
import static java.lang.System.*;

public class DeepMapMain {
    public static void main(String... args) {
        DeepMap<Integer, String> deepMap = new DeepHashMap<>();
        deepMap.put("one/two/three", 1, 2, 3);
        deepMap.put("one/two/four", 1, 2, 4);
        out.println(deepMap.size());
        out.println(deepMap.get(1, 2, 3));       // one/two/three
        out.println(deepMap.at(1, 2, 3));        // String.class
        out.println(deepMap.get(1, 2));          // null
        out.println(deepMap.get(1, 2, 4));       // one/two/four
        out.println(deepMap.get(1, 2, 5));       // null
        out.println(deepMap.get(1, 2, 3, 4));    // null
        out.println(deepMap.at(1, 2));           // DeepHashMap.class
        out.println(deepMap.at(1, 2, 3));        // String.class
        out.println(deepMap.at(1, 2, 3, 4));     // null
        DeepMap<Integer, String> deepSubMap = deepMap.sub(1, 2);
        deepMap.put("one/two", 1, 2);
        out.println(deepMap.size());
        out.println(deepMap.get(1, 2));          // one-two
        out.println(deepMap.get(1, 2, 3));       // null
        out.println(deepSubMap.size());
        out.println(deepSubMap.get(3));          // one/two/three
    }
}
```

Listing 4.66: Testanwendung für die Klasse `DeepHashMap`.

Implementieren Sie das Interface mit einer Klasse `DeepHashMap`.

Kapitel

5 Geschachtelte Klassen

Lernziele

In diesem Kapitel lernen Sie

- dass **statisch geschachtelte Klassen** zwar selbstständig sind, aber dennoch private Elemente miteinander teilen und damit in einer engen Beziehung zueinander stehen.
- dass **innere Klassen** verborgene unveränderliche Referenzen enthalten, mit denen sich abhängige Objekte elegant modellieren lassen.
- dass Objekte **lokaler Klassen** ihre Definitionsumgebung „mitnehmen“, auch wenn sie zum Zeitpunkt des Aufrufs von Methoden schon längst verlassen ist.
- wie **anonyme Klassen** Funktionalität in Objekte kapseln und damit Weg zu funktionaler Programmierung bahnen.
- wie **Lambda-Ausdrücke** von Java 8 die Idee der anonymen Klassen fortsetzen und ihre Anwendung auf elegante Art mit neuen syntaktischen Mitteln vereinfachen.
- dass **Default-Methoden** (ebenfalls Java 8) die bisherige Sprödigkeit von Interfaces aufheben und eine bedeutende Weiterentwicklung der Laufzeitbibliothek ermöglichen.

Eine Java-Quelltextdatei enthält, abgesehen von Package- und Import-Klauseln, eine beliebige Anzahl[1] von Typdefinitionen, das heißt Definitionen von Klassen[2] und Interfaces. Darunter darf höchstens eine Definition `public` sein, deren Name gegebenenfalls an den Dateinamen gekoppelt ist.[3] Diese Klassen und Interfaces

[1] Das schließt null mit ein. Der Java-Compiler akzeptiert auch eine Datei *ohne* Typdefinition, erzeugt daraus aber keinen Bytecode.

[2] Unter den Begriff „Klassen“ fallen auch Aufzählungstypen, abstrakte Klassen, generische Klassen und so weiter.

[3] Andere als `public`-Typen können unabhängig von der Quelltextdatei benannt sein. Ob das hilfreich ist, steht auf einem anderen Blatt.

heißen auch **Toplevel-Klassen** und -Interfaces, weil sie keiner anderen Definition untergeordnet sind, also auf oberster Ebene stehen.[4] Der Löwenanteil der Java-Typdefinitionen sind Toplevel-Klassen und -Interfaces. Java erlaubt aber auch geschachtelte Typdefinitionen. Die Bezeichnungen der verschiedenen Varianten geschachtelter Definitionen sind etwas unübersichtlich, decken sich aber mit [4].

5.1 Statisch geschachtelte Klassen

5.1.1 Definition und Syntax

Klassendefinition in einer anderen Klasse

Die Definition einer **statisch geschachtelten Klasse** steht auf der gleichen Ebene wie Klassenvariablen und statische Methoden, wie im folgenden minimalen Beispiel:

```
class Outer {
   static class Nested {}
}
```

Die Definition von `Nested` ist zwar Teil der Definition von `Outer`, aber das gilt nicht für die Objekte, die unabhängig voneinander sind. Das folgende Codefragment erzeugt ein Objekt der statisch geschachtelten Klasse:[5]

```
Outer.Nested on = new Outer.Nested();
```

Der Typname besteht aus dem Namen der äußeren Klasse und dem Namen der geschachtelten Klasse. Diese Schreibweise steht im Einklang mit der Benennung von Klassenvariablen und statischen Methoden. Der Name `Outer.Nested` kann wie jeder andere Typname verwendet werden.

Zur Laufzeit unabhängige Objekte

`Nested` ist wegen des Modifiers `static` eine *statisch* geschachtelte Klasse. `Nested`-Objekte sind vollkommen selbstständig und sind nicht auf die Existenz von `Outer`-Objekten angewiesen. Um `Nested`-Objekte zu erzeugen ist kein `Outer`-Objekt nötig. So gesehen ist `Nested` nicht mehr als eine Klasse mit einem etwas sonderbaren Namen.

---

[4] Die Package-Hierarchie spielt hier keine Rolle. Packages sorgen nur für ein geregeltes Miteinander mehrerer Definitionen.

[5] Der Ausdruck `Outer.Nested` *kann keine* Klasse `Nested` in einem Package `Outer` bezeichnen, weil alle Typen *und Subpackages* innerhalb eines Packages eindeutige Namen haben müssen. Weil es eine Klassendefinition mit dem Namen `Outer` gibt, kann es kein Subpackage mit dem gleichen Namen geben.

5.1.2 Bedeutung von `private`

Der Nutzen statisch geschachtelter Klassen ergibt sich aus der Wirkung von `private`: Dieser Modifier bezieht sich immer auf die umgebende *Toplevel-Definition*. Anders formuliert: Auch wenn `private` in einer *geschachtelten* Definition steht, hat jedes Mitglied der gesamten Toplevel-Klasse freien Zugriff. Im Einzelnen hat das verschiedene Konsequenzen:

`private` bezogen auf äußerste Klassendefinition

- Methoden der geschachtelten Klasse können auf private Variablen und Methoden der äußeren Klasse zugreifen.
- Auch andersherum stehen private Elemente der geschachtelten Klasse den Methoden der äußeren Klasse zur Verfügung.
- Methoden mehrerer geschachtelter Klassen in der*selben* äußeren Klasse „sehen“ gegenseitig ihre privaten Elemente.[6]

Das folgende Programm verdeutlicht das:

```
import static java.lang.System.*;

public class OuterStatic {
    private static int data = 1;

    private static class Nested {
        private static int data = 2;

        public void method() {
            out.println("Nested.method: " + data);
            out.println("Nested.method: " + OuterStatic.data);
        }
    }

    void method() {
        out.println("OuterStatic.method: " + data);
        out.println("OuterStatic.method: " + Nested.data);
    }

}
```

Listing 5.1: Statisch geschachtelte Klassen.

Zugriff auf private Elemente einer anderen Klasse

Die beiden Methoden mit dem gleichen Namen `method` sind Methoden zweier verschiedener Klassen und haben nichts miteinander zu tun. Redefinition oder Überladen sind nicht im Spiel. Das Gleiche gilt für die privaten Klassenvariablen namens `data`. Unabhängig davon steht den Klassen der Zugriff auf die privaten Elemente der jeweils anderen Klasse frei. Die folgende `main`-Methode zeigt das:

[6] Das gilt auch für statisch *ineinander* geschachtelte Klassen, das heißt statisch geschachtelte Klassen tiefer in statisch geschachtelten Klassen.

```
public static void main(String... args) {
    OuterStatic outer = new OuterStatic();
    outer.method();
    OuterStatic.Nested nested = new OuterStatic.Nested();
    nested.method();
}
```

Listing 5.2: Aufruf der Methoden geschachtelter Klassen.

Sie gibt aus:

```
OuterStatic.method: 1
OuterStatic.method: 2
Nested.method: 2
Nested.method: 1
```

5.1.3 Anwendungsbeispiel: Factory-Methoden

Klasse mit eindeutigen Objekten

Mit statisch geschachtelten Klassen lassen sich eng kooperierende Klassen definieren, die keine Angriffsflächen für unerwünschten Zugriff von außen bieten. Als Beispiel dient eine Klasse `UniqueCharHolder`, deren Objekte nur ein einzelnes Zeichen kapseln. Dabei wird gefordert, dass keine zwei Objekte das gleiche Zeichen enthalten. Die Objekte dieser Klasse können beispielsweise einfach mit den Gleichheitsoperatoren verglichen werden und sind nicht auf `equals` angewiesen.

Um der Forderung nach eindeutigen Objekten nachzukommen, ist Buchführung nötig. Ein geradliniger Ansatz benutzt eine statische Map, die Paare von Zeichen und Objekten speichert. Die Map muss privat bleiben, um Manipulationen auszuschließen. Die ebenfalls statische öffentliche Auskunftsmethode `lookup` überprüft, ob schon ein Objekt mit einem gegebenen Zeichen existiert oder nicht. Sicherheitshalber prüft das auch der Konstruktor der Klasse nach und wirft gegebenenfalls eine Exception.

```
import java.util.*;

public class UniqueCharHolder0 {
    private final char value;

    private final static Map<Character, UniqueCharHolder0> objects = new HashMap<>();

    public UniqueCharHolder0(char value) {
        if(objects.containsKey(value))
            throw new IllegalArgumentException("object exists");
        objects.put(value, this);
        this.value = value;
    }
```

```
    public static UniqueCharHolder0 lookup(char value) {
        return objects.containsKey(value)? objects.get(value): null;
    }

}
```

Listing 5.3: Klasse, die ihre Objekte überwacht.

Objekte erzeugen nach Bedarf

Die folgende `main`-Methode erzeugt einige Objekte aus den Anfangsbuchstaben der Kommandozeilenargumente. Sie versucht zuerst, ein Objekt mit `lookup` zu finden. Wenn es noch keines gibt (`lookup` liefert `null`), ruft `main` den Konstruktor auf, der ein neues Objekt erzeugt und gleichzeitig aufzeichnet.

```
public static void main(String[] args) {
    for(String arg: args) {
        UniqueCharHolder0 holder = UniqueCharHolder0.lookup(arg.charAt(0));
        if(holder == null)
            holder = new UniqueCharHolder0(arg.charAt(0));
        System.out.println(holder);
    }
}
```

Listing 5.4: Test der Klasse mit überwachten Objekten.

Ein Aufruf zeigt, dass das Programm wie gewünscht arbeitet:

```
$ java UniqueCharHolder0 a b b a
UniqueCharHolder0@e1fb6c
UniqueCharHolder0@41825e
UniqueCharHolder0@41825e
UniqueCharHolder0@e1fb6c
```

Statische Factory-Methode kapselt Objektverwaltung

Diese Implementierung hat Nachteile: Sie belastet die Anwendung mit der Objektverwaltung und sie straft einen unzulässigen Konstruktoraufruf mit einem Laufzeitfehler ab.[7] Diese Mängel beseitigt eine statische Factory-Methode, wobei gleichzeitig der `UniqueCharHolder`-Konstruktor mit `private` dem allgemeinen Zugriff entzogen wird:

```
import java.io.*;
import java.util.*;

public class UniqueCharHolder1 {
```

[7] Laufzeitfehler sind weit teurer als Compilerfehler, weil sie erst spät auftreten.

```
    private final char value;

    private static Map<Character, UniqueCharHolder1> charObjects = new HashMap<>();

    private UniqueCharHolder1(char value) {
        this.value = value;
    }

    public static UniqueCharHolder1 make(char value) {
        if(!charObjects.containsKey(value))
            charObjects.put(value, new UniqueCharHolder1(value));
        return charObjects.get(value);
    }

}
```

Listing 5.5: Klasse mit privatem Konstruktor, deren Objekte von einer Factory-Methode geliefert werden.

Die Anwendung muss sich jetzt nicht mehr um die Objektverwaltung kümmern, kann aber keinen Konstruktor mehr aufrufen und muss stattdessen die Methode `make` bemühen:

```
public static void main(String[] args) throws IOException {
    for(String arg: args) {
        UniqueCharHolder1 holder = UniqueCharHolder1.make(arg.charAt(0));
        System.out.println(holder);
    }
}
```

Listing 5.6: Aufruf einer Factory-Methode statt eines Konstruktors.

Objektverwaltung in einer eigenen Klasse

Von außen betrachtet ist das Problem gelöst. Unbefriedigend bleibt aber das Durcheinander der eigentlichen Klassenlogik und der Objektverwaltung im Code der Klassendefinition. Diese zwei Aufgaben haben nichts miteinander zu tun und sollten getrennt werden. Dazu wird die Factory-Methode samt der Map in eine eigene Klasse verschoben. Diese Klasse hätte allerdings keinen Zugriff mehr auf den privaten Konstruktor. Eine statisch geschachtelte Klasse löst das Problem, weil sie den privaten Konstruktor weiterhin ungehindert aufrufen kann:

```
import java.util.*;

public class UniqueCharHolder {
    private final char value;

    private UniqueCharHolder(char value) {
        this.value = value;
    }
```

```
    public static class Maker {
        private static Map<Character, UniqueCharHolder> charObjects = new HashMap<>();

        public UniqueCharHolder make(char value) {
            if(!charObjects.containsKey(value))
                charObjects.put(value, new UniqueCharHolder(value));
            return charObjects.get(value);
        }
    }

}
```

Listing 5.7: Klasse mit statisch geschachtelter Factory-Klasse.

Die Anwendung holt sich zunächst ein Objekt der statisch geschachtelten Klasse und ruft dann dessen Factory-Methode auf:

```
public static void main(String[] args) {
    UniqueCharHolder.Maker maker = new UniqueCharHolder.Maker();
    for(String arg: args) {
        UniqueCharHolder holder = maker.make(arg.charAt(0));
        System.out.println(holder);
    }
}
```

Listing 5.8: Anwendung erzeugt Objekte über Factory.

Ableiten einer statisch geschachtelten Klasse

Statisch geschachtelte Klassen können wie andere Klassen verwendet werden. Die folgende Klasse `EagerMaker` ist von `Maker` abgeleitet und redefiniert die Methode `make` so, dass sie „auf Verdacht“ auch gleich zwei Objekte mit benachbarten Zeichen erzeugt. `EagerMaker` ist eine ganz gewöhnliche Klasse und hat *keinen* Zugriff auf die privaten Elemente von `UniqueCharHolder`.

```
public class EagerMaker extends UniqueCharHolder.Maker {
    public UniqueCharHolder make(char value) {
        super.make((char)(value - 1));
        super.make((char)(value + 1));
        return super.make(value);
    }
}
```

Listing 5.9: Statisch geschachtelte Klasse als Basisklasse für Ableitung.

Lösung mit Packages

Eine vergleichbare Lösung ist mit einem Package möglich, das nur `UniqueCharHolder` und `Maker` enthält und das dem einzigen Zweck dient, eine gemeinsame „Außengrenze“ um diese beiden Klassen zu ziehen. Dieser Ansatz ist aber weniger flexibel, weil das Package keine anderen Definitionen enthalten dürfte und mit weiteren Klassen nur noch auf der Ebene `public` kommunizieren könnte.

Löchriger Schutz bei Packages

▶ Packages bieten keinen sehr zuverlässigen Schutz gegenüber „zudringlichen Typen“. Ein Package entspricht einem Directory im Filesystem. Alle Typdefinitionen in diesem Directory gehören zum selben Package. Allerdings können an verschiedenen Punkten des Classpath Directories mit dem gleichen Namen angelegt werden. Zu einem Package gehören die Typdefinitionen aus *allen* diesen Directories.

Wenn beispielsweise das aktuelle Arbeitsdirectory (`.`) im Classpath enthalten ist, kann man den Quelltext

```
package java.lang;
public class Foo {}
```

der Klasse `Foo` im Directory `java/lang` speichern und dort übersetzen:

```
$ javac java/lang/Foo.java
```

`Foo` wird damit Mitglied von `java.lang` und steht global ohne Import zur Verfügung, wie alle Elemente von `java.lang`.

Eine benutzerdefinierte Klasse `Foo` in die Prominenz von `java.lang` einzureihen, grenzt natürlich an Hochstapelei. Es gibt aber kein technisches Hindernis, sich nachträglich und ungebeten in ein Package einzumischen. Mit statisch geschachtelten Klassen ist das ohne Manipulationen am Quelltext des Gastgebers nicht möglich. Statisch geschachtelte Klassen bleiben definitiv unter sich. ◀

5.1.4 Bytecode-Dateien

Namen der Bytecode-Dateien geschachtelter Klassen

Der Java-Compiler erzeugt aus Typdefinitionen Bytecode-Dateien, deren Namensrumpf mit dem Typnamen übereinstimmt. Das gilt auch für geschachtelte Typen, die in eigene Bytecode-Dateien übersetzt werden. Um Konflikten mit den Namensregeln des Filesystems aus dem Weg zu gehen, ersetzt der Compiler Punkte in Typnamen durch Dollar-Zeichen.[8]

Der Compiler übersetzt `UniqueCharHolder` (Listing 5.7) zum Beispiel in die beiden folgenden Dateien:[9]

[8] Wenn es bereits eine andere Klasse mit genau dem Namen gibt, den der Compiler durch Zeichenersetzung erzeugt, dann stoppt der Compiler mit einem Fehler `duplicate class`. Benutzerdefinierte Typnamen sollten keine Dollar-Zeichen und Unterstriche enthalten, auch wenn das syntaktisch zulässig ist.

[9] Obwohl das Dollar-Zeichen in Java-Identifiern zulässig ist, wird `UniqueCharHolder$Maker` im Quelltext nicht als alternative Schreibweise für `UniqueCharHolder.Maker` akzeptiert.

```
UniqueCharHolder.class
UniqueCharHolder$Maker.class
```

5.1.5 Statisch geschachtelte Interfaces

In Klassen geschachtelte Interfaces

Abgesehen von Klassen können auch Interfaces im Kontext einer Klasse definiert werden. Die Spielregeln sind die gleichen wie für Klassen. Geschachtelte Interfaces haben Zugriff auf private Elemente der äußeren Klasse, wie im folgenden Beispiel:

```
class Outer {
   private static int a = 1;
   interface IfNested {    // implizit static
       int b = a + 1;      // implizit public static final
   }
}
```

Geschachtelte Interfaces sind automatisch statisch, auch wenn der Modifier `static` fehlt. Jede Klasse mit Zugriff auf ein solches Interface kann es implementieren, ähnlich wie `EagerMaker` (Listing 5.9) eine statisch geschachtelte Klasse ableitet.

Formal lassen sich auch innerhalb von Interfaces statisch geschachtelte Klassen und weitere Interfaces definieren. Nachdem aber alle Elemente des äußeren Interface ohnedies `public` sind, eröffnen sich keine besonderen Zugriffsmöglichkeiten.

5.2 Innere Klassen

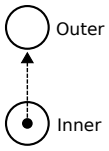
Klassendefinitionen als Elemente von Klassen

Innere Klassen werden wie statisch geschachtelte Klassen definiert, allerdings ohne den Modifier `static`, wie das folgende minimale Beispiel zeigt:

```
class Outer {
   class Inner {}
}
```

Bindung an Objekt der äußeren Klasse

Wie Objektvariablen und Methoden von `Outer` ist auch `Inner` ein gleichrangiges Mitglied der Klasse und genießt freien Zugriff auf alle Mitbewohner. Das bedeutet zum Beispiel, dass den Methoden von `Inner` die Objektvariablen von `Outer` ohne weitere Maßnahmen zur Verfügung stehen. Das bedeutet aber auch, dass ein isoliertes `Inner`-Objekt nicht existieren kann, weil es sonst keine `Outer`-Objektvariablen gäbe! Ein `Inner`-Objekt ist auf Lebenszeit mit genau einem `Outer`-Objekt verbunden.



Versteckte, unveränderliche Referenz

Diese Beziehung ist unveränderlich und manifestiert sich in einer unsichtbaren `final`-Objektvariablen vom Typ `Outer` in jedem `Inner`-Objekt. Diese verborgene Referenz kann nicht beeinflusst werden. In Wahrheit sieht die Definition von `Inner` also etwa folgendermaßen aus:

```
class Outer {
   class Inner
   {
        private final Outer myOuter;
       Inner( Outer myOuter)
       {
            this.myOuter = myOuter;
       }
   }
}
```

Erzeugen innerer Objekte nur mit äußerem Objekt

`Inner` und `Outer` sind getrennte Klassen, deren Objekte einzeln erzeugt werden müssen. Um ein `Inner`-Objekt zu produzieren, muss allerdings bereits ein `Outer`-Objekt existieren, auf das im Konstruktor die verborgene Referenz angelegt wird. Syntaktisch folgt der Operator `new` dem `Outer`-Bezugsobjekt:

```
Outer o = new Outer();
Outer.Inner oi = o.new Inner();
```

▶ Der Java-Disassembler macht diese Konstruktion sichtbar:

```
$ javap -p -c Outer$Inner.class
Compiled from "Outer.java"
01 public class Outer$Inner {
02   final Outer this$0;
03
04   public Outer$Inner(Outer);
05     Code:
06       0: aload_0
07       1: aload_1
08       2: putfield      #1 // Field this$0:LOuter;
09       5: aload_0
10       6: invokespecial #2 // Method java/lang/Object."<init>":()V
11       9: return
12 }
```

Zum einen findet man in Zeile 2 die Definition der verborgenen Objektvariablen namens `this$0` vom Typ `Outer`. Zum anderen erkennt man einen zusätzlich generierten `Outer`-Parameter des Konstruktors (Zeile 4), mit dem die verborgene Referenz initialisiert wird (Zeilen 7/8). ◀

Beziehung zwischen äußerer und innerer Klasse

Eine innere und eine äußere Klasse stehen in einer besonderen Beziehung, ebenso wie eine Basisklasse und eine abgeleitete Klasse. Dabei gibt es grundsätzliche Unterschiede:

- Eine abgeleitete Klasse ist kompatibel zur Basisklasse, eine innere Klasse aber nicht zur äußeren Klasse (Compiler).
- Eine innere Klassen kann frei auf private Elemente der äußeren Klasse zugreifen, einer abgeleiteten Klasse steht das nicht zu (Compiler).
- Aus einer Basisklasse und einer abgeleiteten Klasse entsteht ein einziges Objekt mit den Bausteinen beider Klassen, Objekte einer inneren und einer äußeren Klasse liegen getrennt, wenn auch mit einer Referenz gekoppelt (Laufzeit).

5.2.1 Anwendungsbeispiel: Iterierbare Strings

Anwendungszweck: abhängige Objekte

Innere Klassen haben trotz der syntaktischen Ähnlichkeit andere Anwendungen als statisch geschachtelte Klassen. Sie eignen sich für abhängige Objekte, die nur im Kontext eines anderen Objekts sinnvoll sind.

Als Beispiel dient eine Klasse, die Strings für *foreach*-Schleifen zugänglich macht. Strings implementieren selbst nicht das dazu nötige Interface `Iterable<Character>`, deshalb übernimmt das die Klasse `IterableString`. Diese Klasse kapselt einen String und definiert die Methode `iterator`, die das Interface `Iterable` verlangt. Die Methode `iterator` liefert ein `Iterator`-kompatibles Objekt, in dem der eigentliche Aufwand steckt:

```
import java.util.*;

public class IterableString0 implements Iterable<Character> {
    private final String string;

    public IterableString0(String string) {
        this.string = string;
    }

    public Iterator<Character> iterator() {
        return new StringIterator(string);
    }
}
```

Listing 5.10: Klasse, die die Zeichen eines Strings in *foreach*-Schleifen verfügbar macht.

Lösung mit getrennten Klassen

Ein geradliniger Ansatz definiert dazu eine getrennte Klasse `StringIterator`, die das Interface `Iterator<Character>` implementiert und die drei dafür erforderlichen Methoden `hasNext`, `next` und `remove`[10] implementiert:

```
import java.util.*;

public class StringIterator implements Iterator<Character> {
    private final String string;

    private int next = 0;

    public StringIterator(String string) {
        this.string = string;
    }

    public boolean hasNext() {
        return next < string.length();
    }

    public Character next() {
        return string.charAt(next++);
    }

    public void remove() {
        throw new UnsupportedOperationException("Not supported.");
    }
}
```

Listing 5.11: Iterator über die Zeichen eines Strings als externe Hilfsklasse für ein `Iterable`-Objekt.

Einfachere Lösung mit einer inneren Klasse

Die Lösung funktioniert zwar, lässt sich aber vereinfachen. Dazu muss nur die Definition von `StringIterator` in `IterableString` verschoben werden. Der `StringIterator` hat jetzt freien Zugriff auf die private Objektvariable `string` der äußeren Klasse. Deshalb kann die Objektvariable `string` in der inneren Klasse wegfallen und infolgedessen auch der Konstruktor:

```
import java.util.*;

public class IterableString implements Iterable<Character> {
    private final String string;

    public IterableString(String string) {
        this.string = string;
    }

    private class StringIterator implements Iterator<Character> {
        private int next = 0;
```

[10] Strings sind unveränderlich. `remove` darf daher nicht aufgerufen werden und wirft eine `UnsupportedOperationException`.

```
        public boolean hasNext() {
            return next < string.length();
        }

        public Character next() {
            return string.charAt(next++);
        }

        public void remove() {
            throw new UnsupportedOperationException("Not supported.");
        }
    }

    public Iterator<Character> iterator() {
        return new StringIterator();
    }
}
```

Listing 5.12: Innere `Iterator`-Klasse als private Hilfsklasse.

Private Klasse mit öffentlichen Methoden

Obwohl `StringIterator` selbst privat ist, definiert die Klasse öffentliche Methoden. Das ist in Ordnung, weil der Name der inneren Klasse nur in `IterableString` verwendet wird. Außerhalb ist der Name `StringIterator` nicht sichtbar und wird auch nicht gebraucht.

Mit dieser Klasse lässt sich bequem über die Zeichen eines Strings iterieren:

```
public static void main(String... args) {
    for(char chr: new IterableString(args[0]))
        System.out.println(chr);
}
```

Listing 5.13: *foreach*-Schleife über die Einzelzeichen eines Strings.

5.3 Lokale Klassen

Klassendefinition im Rumpf einer Methode

Lokale Variablen sind allgegenwärtig: Sie werden in Methodenrümpfen auf der Ebene von Anweisungen definiert und gelten von der Definition bis zum Ende des Definitionsblock. Auf der gleichen Ebene stehen **lokale Klassen.** Ihr Gültigkeitsbereich unterliegt den gleichen Regeln wie Variablen.[11]

```
class Something {
   void method() {
       class Local {}
```

[11] Das bedeutet zum Beispiel, dass eine Klasse, die lokal im Rumpf einer Schleife definiert ist, nur im Schleifenrumpf existiert.

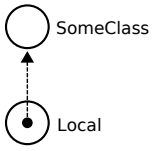
```
        Local l = new Local();
    }
}
```

Unsichtbar außerhalb des Definitionsblocks

Ebenso wie innere Klassen können lokale Klassen alle privaten Elemente der Klasse, in der die Methode definiert ist, erreichen.[12] Andersherum gilt das nicht: Die lokale Klasse ist nur innerhalb des Definitionsblocks bekannt. Außerhalb des Blocks existiert sie nicht. Die Frage nach Zugriffsmöglichkeiten auf Bausteine stellt sich überhaupt nicht.

Verborgene Referenz auf Zielobjekt

Jedes Objekt einer lokalen Klasse trägt eine verborgene Referenz auf ein Objekt der umgebenden Klasse in sich. Die Situation ist die gleiche wie bei inneren Klassen.[13]



Rückgabe eines Objekts einer lokalen Klasse

Auf den ersten Blick scheint es keine Möglichkeit zu geben, wie Objekte einer lokalen Klasse den Methodenrumpf verlassen können. Schließlich ist in der Außenwelt nicht einmal der Name der Klasse bekannt! Eine Definition in der Art

```
class Something {
    Local method() {     // wird nicht übersetzt
       class Local {}
       return new Local();
    }
```

verweigert der Compiler, weil der Typ `Local` nur *im* Rumpf der Methode definiert ist und nicht als Ergebnistyp verwendet werden kann. Das Bild ändert sich, wenn die Kompatibilität abgeleiteter Klassen genutzt wird. Das Programm wird übersetzt, wenn die Methode beispielsweise `Object` zurückgibt, weil `Local` ebenso zu `Object` kompatibel ist wie alle Referenztypen. Das folgende Programm wird übersetzt:

```
class Something {
    Object method() {
        class Local {
            public String toString() {
```

[12] Dazu kommen noch die lokalen Variablen am Ort der Definition. Diese unscheinbare Möglichkeit hat weitreichende Konsequenzen, die weiter unten genauer untersucht werden.

[13] Der Vollständigkeit halber sei angemerkt, dass in statischen Methoden „statisch lokale Klassen“ definiert werden können. Diese tragen *keine* verborgene Referenz auf ein Objekt.

```
                return "Local";
            }
        }
        return new Local();
    }

    public static void main(String[] args) {
        Something something = new Something();
        Object x = something.method();
        System.out.println(x);
    }
}
```

Listing 5.14: Rückgabe einer Referenz auf ein Objekt einer lokalen Klasse als `Object`.

Es gibt aus:

```
$ java Something
Local
```

Bezug auf die Definitionsumgebung in der Aufrufumgebung

Lokale Klassen haben Zugriff auf die privaten Elemente der umgebenden Klassendefinition und außerdem *auf lokale Variablen* (einschließlich der Methodenparameter), die am Ort ihrer Definition gelten. Das wirft eine interessante Frage auf: Eine Methode einer lokalen Klasse, die auf Variablen der Definitionsumgebung Bezug nimmt, kann zu einem späteren Zeitpunkt in einer ganz anderen Umgebung aufgerufen werden. Dort existiert die Definitionsumgebung aber längst nicht mehr und folglich auch keine lokalen Variablen dieser Umgebung! Wie kann eine Methode Variablen einer Umgebung verwenden, die es nicht mehr gibt?

Das nächste Programm illustriert das Problem:

```
001 class SomethingElse {
002     Object method(final int i) {
003         class Local {
004             public String toString() {
005                 return "Local, i = " + i;
006             }
007         }
008         return new Local();
009     }
010
011     public static void main(String[] args) {
012         SomethingElse s = new SomethingElse();
013         Object x = s.method(23);
014         System.out.println(x.toString());
015     }
016 }
```

Listing 5.15: Verwendung eines Parameters nach Rückkehr der Methode.

Zugriff auf Variablen nach formalem Ende der Lebenszeit

Entscheidend sind beiden Zeilen 013 und 014:

- In Zeile 013 wird `method` mit dem Argument 23 aufgerufen und führt den Rumpf (Zeilen 002–009) aus. Für die Dauer dieses Aufrufs existiert der Parameter `i` mit dem Wert 23. Bei der Rückkehr von `method` verschwindet der Parameter.
- In Zeile 014 wird `toString` aufgerufen. Diese Methode braucht den Parameter `i`, um ihn auszugeben. Zu diesem Zeitpunkt ist der Aufruf von `method` aber schon beendet und der Parameter `i` *existiert nicht mehr!*

Dennoch funktioniert das Programm:

```
$ java SomethingElse
Local, i = 23
```

Implizites Sichern lokaler Variablen

Das ist möglich, weil der Compiler die Definition von `Local` automatisch um eine verborgene, unveränderliche Objektvariable erweitert, die den Wert von `i` aufbewahrt. Die „eingefangene“ lokale Variable *muss* in der Definitionsumgebung mit dem Modifier `final` geschützt sein, weil sie andernfalls nach der Übernahme in die Klassendefinition noch verändert werden könnte. Ohne `final` übersetzt der Compiler das Programm nicht.[14] Der Code von `method` sieht also in Wahrheit etwa folgendermaßen aus:[15]

```
Object method(final int i) {
    class Local {
         private final int i;
         Local(int i) {
             this.i = i;
         }
        public String toString() {
            return "Local, i = " + i;
        }
    }
    return new  Local(i);
}
```

Zeitliche Trennung von Definitions- und Ablaufumgebung

Um das Problem noch einmal klar herauszustellen: Objekte einer lokalen Klasse werden in einer anderen Umgebung (Definitionsumgebung) erzeugt als in der Umgebung (Ablaufumgebung), in der die Methoden aufgerufen werden. Trotzdem

[14] Die Wirkung von `final` hat ihre Grenzen. Sie schützt nur die Variable vor Änderungen, aber im Fall von Referenzvariablen nicht das referenzierte Objekt. Nachträgliche Modifikationen an einem solchen Objekt ziehen Aliasing-Effekte nach sich.

[15] In dieser Skizze fehlt die zweite, ebenfalls verborgene Referenz auf das Objekt der umgebenden Klasse zur besseren Lesbarkeit.

müssen Variablen aus der Definitionsumgebung in der Ablaufumgebung zur Verfügung stehen.

Closures bewahren Definitionsumgebung auf

Dazu müssen die Objekte die Definitionsumgebung mit sich führen, sie „schließen sie ein“. Eine solche Konstruktion wird als **Closure** bezeichnet. Eine Closure besteht im Allgemeinen aus Code (im Fall lokaler Klassen aus Methoden) und der Umgebung (verborgene Objektvariablen), in der der Code irgendwann ausgeführt wird.

Dateinamen übersetzter lokaler Klassen

Der Compiler übersetzt lokale Klassen in isolierte Bytecode-Dateien. Die Namen dieser Dateien setzen sich aus dem Namen der umgebenden Klasse, einer laufenden Nummer und dem Namen der lokalen Klasse zusammen. Die im Quelltext erste lokale Klasse erhält die Nummer 1, die nächste 2 und so weiter. Aus dem Code von `SomethingElse` (Listing 5.15) erzeugt der Compiler die beiden folgenden Dateien:

```
SomethingElse.class
SomethingElse$1Local.class
```

Lokale Klassen werden nicht oft gebraucht. Eine Anwendungen findet sich im Programm `PermutationCounter` (Listing 4.44). Eng verwandt mit lokalen Klassen sind „anonyme Klassen“, die im nächsten Abschnitt besprochen werden. Im Gegensatz zu lokalen Klassen spielen anonyme Klassen eine größere Rolle.

5.4 Anonyme Klassen

Klassendefinition samt Initialisierung als Ausdruck

„Anonyme Klassen“ sind Klassendefinitionen im Kontext von Ausdrücken. Allerdings hat ein Ausdruck einen Wert, eine Klassendefinition aber nicht. Eine anonyme Klasse wird daher nicht nur definiert, sondern auch sofort, das heißt „an Ort und Stelle“, instanziiert. Das Objekt, das dabei erzeugt wird, ist gleichzeitig der Wert des Ausdrucks.

Namenlose abgeleitete Klasse

Anonyme Klassen haben keinen Namen und beziehen sich immer auf einen Basistyp, das heißt auf ein Interface oder eine Basisklasse. Das neu erzeugte Objekt ist kompatibel zum Basistyp, sein konkreter Typ bleibt aber im Dunkeln.

Syntaktisch repräsentiert der folgende Ausdruck ein Objekt einer namenlosen, zu *Type* kompatiblen Klasse:

```
new Type(args) {...}
```

Die Argumente *args* gehen direkt an den *Type*-Konstruktor.[16] Der Rumpf in den geschweiften Klammern implementiert oder redefiniert *Type*-Methoden.[17]

Ähnlichkeit lokale und anonyme Klassen

Eine anonyme Klasse hat Eigenschaften innerer und lokaler Klassen. Ebenso wie bei

- inneren Klassen stehen alle privaten Elemente der umgebenden Toplevel-Klassendefinition zur Verfügung.
- lokalen Klassen bildet eine anonyme Klasse eine Closure über der Umgebung am Punkt der Definition. Das schließt insbesondere die lokalen `final`-Variablen ein.

Einschränkungen anonymer Klassen

Gegenüber lokalen Klassen unterliegen anonyme Klassen einigen Einschränkungen:

- Definition und Instanziierung fallen untrennbar zusammen.
- Sie haben keinen Namen.
- Die Konstruktoren werden automatisch aus den Basisklassenkonstruktoren generiert mit dem Rumpf

  ```
  super(args);
  ```

  Bei Interfaces als Basistyp gibt es nur einen leeren Default-Konstruktor.
- Zusätzliche öffentliche Methoden über die des Basistyps hinaus sind sinnlos, weil sie von außen nicht angesprochen werden könnten.
- Anonyme Klassen sind immer `final` und können daher weder abgeleitet werden noch abstrakt sein.

5.4.1 Anwendungsbeispiele

Anwendungsmöglichkeiten für anonyme Klassen

Anonyme Klassen erlauben oft einfachere Lösungen als getrennte, explizit definierte Klassen. Sie kommen dann in Betracht, wenn

- nur ein einziges Objekt gebraucht wird,
- das Objekt ein Interface implementieren oder eine Basisklasse ableiten soll,
- nur wenige Methoden zu definieren oder zu redefinieren sind und
- die wenigen Methoden nicht allzu komplex sind.

Im Folgenden werden einige typische Anwendungen gezeigt.

[16] Wenn *Type* ein Interface ist, bleibt die Argumentliste leer.

[17] Ein leerer Rumpf ist sinnlos. In diesem Fall leistet die anonyme Klasse nicht mehr als *Type* und hätte keine Daseinsberechtigung.

`Comparator`-Objekte

Sortierkriterien von Collections

Die Collection-Klassen `TreeSet` und `TreeMap` erhalten unter ihren Elementen eine Reihenfolge aufrecht, die sich in der Voreinstellung aus der „natürlichen Ordnung“ ergibt. Die natürliche Ordnung stützt sich auf die Methode `compareTo` des Interface `Comparable`, das der Elementtyp implementieren muss.[18] Das folgende Programm fügt die Kommandozeilenargumente in ein `TreeSet` ein und gibt es dann aus:

```
import java.util.*;

public class NaturalStringOrdering {
    public static void main(String... args) {
        TreeSet<String> set = new TreeSet<>();
        for(String arg: args)
            set.add(arg);
        System.out.println(set);
    }
}
```

Listing 5.16: Einfügen der Kommandozeilenargumente in eine sortierte Collection.

Der Aufruf sortiert erwartungsgemäß die Kommandozeilenargumente alphabetisch:

```
$ java NaturalStringOrdering abra ka dabra sim sala bim
[abra, bim, dabra, ka, sala, sim]
```

Externe `Comparator`-Definition

Wahlweise kann den Konstruktoren der Collection-Klassen auch ein `Comparator`-Objekt mitgegeben werden, dessen Methode `compare` dann die Ordnung der Element regelt.[19] Der folgende Comparator ordnet Strings nach steigender Länge:

```
import java.util.*;

public class StringLengthComparator implements Comparator<String> {
    public int compare(String s0, String s1) {
        return s0.length() - s1.length();
    }
}
```

Listing 5.17: String-Vergleicher nach steigender Länge.

[18] `TreeSet` und `TreeMap` verwenden weder `equals` noch `hashCode`.

[19] Ein solches `TreeSet` akzeptiert *jeden* Elementtyp. Er muss insbesondere nicht mehr zu `Comparable` kompatibel sein.

Er kann zur Initialisierung eines `TreeSet` verwendet werden:

```
TreeSet<String> ts = new TreeSet<>(new StringLengthComparator());
```

Das Programm gibt jetzt aus:[20]

```
$ java CustomStringOrdering abra ka dabra sim sala bim
[ka, sim, abra, dabra]
```

Anonyme `Comparator`-Definition

Die explizit definierte `Comparator`-Klasse kann durch ein anonymes `Comparator`-Objekt ersetzt werden, wie in der folgenden Definition:[21]

```
TreeSet<String> set = new TreeSet<>(new Comparator<String>() {
    public int compare(String s0, String s1) {
        return s0.length() - s1.length();
    }
});
```

Listing 5.18: Collection mit anonymem Vergleicher-Objekt als Sortierkriterium.

`Thread`-Klassen

Anonyme `Thread`-Klasse

Threads definieren parallel ablaufenden Code in Klassen, die die Methode `run` der Basisklasse `Thread` redefinieren (siehe Kapitel 6). Dabei ist die konkrete abgeleitete Klasse oft weit weniger interessant als der Rumpf der Methode `run`. Für einfache Thread-Klassen eignen sich anonyme Klassen.

Das folgende Programm erzeugt und startet für jedes Kommandozeilenargument einen neuen Thread, der das Kommandozeilenargument hundertmal ausgibt. Die Schleifenvariable `arg` muss `final` definiert sein, weil die anonyme Klasse darauf zugreift.[22]

---

[20] Von den sechs Kommandozeilenargumenten erscheinen nur vier in der Ausgabe. Der Comparator sortiert die Strings nach ihrer Länge und bewertet folglich gleich lange Strings als „gleich“. Ein Set speichert keine Duplikate und ignoriert daher Strings, die gemäß Comparator bereits im Set enthalten sind.

[21] Die Type-Inference des Compilers von Oracle Java SE 7 reicht nicht aus, um das Typargument der Definition einer generischen anonymen Klasse zu erschließen. Statt des Diamond-Operators muss das Typargument explizit genannt werden.

[22] Bei einer *foreach*-Schleife ist das möglich, weil in jedem Schleifendurchgang eine neue Inkarnation der Schleifenvariablen angelegt wird. Die Laufvariable einer regulären `for`-Schleife existiert nur einmal und kann nicht `final` sein.

```
public class ParallelPrinter {
    public static void main(String... args) {
        for(final String arg: args)
            new Thread() {
            public void run() {
                for(int i = 0; i < 100; i++) {
                    System.out.print(arg);
                    Thread.yield();
                }
            }
        } .start();
    }
}
```

Listing 5.19: Anonyme `Thread`-Klasse mit `run`-Methode, die einen String mehrmals ausgibt.

Der Programmstart produziert eine wilde Mischung aller Ausgaben, wie es typisch für unsynchronisierte Threads ist.

```
$ java ParallelPrinter a b c d e f g h
aaaaaaaaaaaaaaaaaaaabaabababababababababababababababababababababababababaaaaaaaa
aababababaaababababababababababababababababababdababdabdabdabdabddbadbadbadbadbadbaccccccc
ccccccccccaaaaaaaabbbbbdbdbdbdbdbdebdbdbedbdbedbedbedbedbedbedbedbedbedbedbedbed
bedbedbedbccccccccccccccccccccbcdefedbcedcbdecbdecbdecbdecbdecdecedcedcedcecdecdecdec
decdecdcecdecdecdecdecdecdecdefdecdgdffddddffgcefgggghdfhcefhgdfhcefhgdfhcefhgdfhecfh
gdfhcefhgddfhcefhgdfhecfhgdfhcefhgdfffffffffffffffffffffffffffffffffffffffffffff
fffffeccccecececececececececfffhgdddddddddgdgdgddgdgdghgfggggceeeecccccccccccccgfhdfge
fhdfgefdhffffffffffffgeghgeghgegghgeghgegghggeghgeghgeeeeeeeeeeeeeeeeeeeeghhhhhhhhh
hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhgggggghgggggggggggggggggggggggggggggggg
gggggghhhhhhhhhhhhhhhhhg
$
```

Die Ausgabe ist im Detail nicht vorhersagbar und nicht reproduzierbar. Sie sieht bei jedem Programmaufruf anders aus.

Iterierbare Strings

Die Klasse `IterableString` (Listing 5.12) enthält eine innere Klasse `StringIterator`, die

Anonyme `Iterator`-Implementierung

- an einer einzigen Stelle benutzt wird, nämlich in der Methode `iterator`,
- nur einen automatisch definierten Default-Konstruktor braucht,
- das Interface `Iterator<Character>` implementiert,
- nur die Methoden des Interface definiert und
- nicht weiter abgeleitet wird.

Diese Kriterien sprechen für eine anonyme Klasse, wie im folgenden Programm:

```
import java.util.*;

public class IterableStringAnonymous implements Iterable<Character> {
    private final String string;

    public IterableStringAnonymous(String string) {
        this.string = string;
    }

    public Iterator<Character> iterator() {
        return new Iterator<Character>() {
            // Rumpf wie StringIterator ...
        };
    }
}
```

Listing 5.20: Anonyme `Iterator`-Klasse zur Implementierung des `Iterable`-Interface.

Unübersichtlicher Code wegen zu komplexer anonymer Klasse

Die Komplexität der anonymen Klasse liegt in diesem Beispiel an der Grenze des Sinnvollen. Rein technisch und unter Missachtung der Übersichtlichkeit kann sogar die ganze `Iterable`-Klasse durch eine anonyme Klasse ersetzt werden. Dieses Beispiel ist allerdings nicht nachahmenswert und dient nur zur Illustration geschachtelter anonymer Klassen:

```
public static void main(String... args) {
    final String string = args[0];
    for(char chr: new Iterable<Character>() {
    public Iterator<Character> iterator() {
            return new Iterator<Character>() {
                // Rumpf wie StringIterator ...
            };
        }
    })
    System.out.println(chr);
}
```

Listing 5.21: Geschachtelte anonyme Klassen.

Anonyme Klassen sind eine nützliche Konstruktion. Weitere Anwendungsbeispiele finden Sie in `FindShortFiles`, `PermutationPrinter` (Listing 4.42), `ParallelFibonacci` und `ClocktimeSunServer`.

5.5 Lambda-Ausdrücke (Java 8)

Vereinfachte Form anonymer Klassen

Java-Version 8 führt sogenannte **Lambda-Ausdrücke** ein, die anonyme Klassen

in den meisten Fällen ersetzen und zu einfacherem Code führen. Die Lambda-Ausdrücke von Java 8 sind allerdings keine vollwertigen *First-class*-Funktionen, wie sie andere Programmiersprachen, wie zum Beispiel Javascript, Scheme und Haskell bieten. Ziele beim Entwurf der Lambda-Ausdrücke waren problemlose Anwendbarkeit, geradlinige Arbeitsweise und gute Verträglichkeit mit bestehendem Java-Code. Einschränkungen gegenüber funktionalen Sprachen wurden deshalb bewusst in Kauf genommen.

5.5.1 Funktionsinterfaces

Lambda-Ausdrücke hängen direkt mit **Funktionsinterfaces** zusammen. Ein Funktionsinterface ist ein normales Java-Interface mit *genau einer* Methode. Interfaces ganz ohne[23] oder mit mehreren Methoden sind keine Funktionsinterfaces.[24]

Funktionsinterface: eine einzige Methode

Ein Beispiel eines Funktionsinterface ist

```
public interface Code {
    void run();
}
```

Listing 5.22: Ein Stück Code ohne Argumente und ohne Ergebnis, das allenfalls durch Seiteneffekte wirkt.

Die Interfaces `Runnable`, `Comparator`, `Callable` und `Iterable` aus der Bibliothek sind ebenfalls Funktionsinterfaces, nicht aber `FileVisitor` (vier Methoden), `Iterator` (drei Methoden) und `Enumeration` (zwei Methoden).

Öffentliche `Object`-Methoden zählen dabei nicht mit, auch wenn sie ein Interface ausdrücklich ausweist, weil ohnedies jedes Objekt eine triviale Implementierung erbt. Beispielsweise ist `Comparator` ein Funktionsinterface, obwohl in der Definition textuell die *zwei* Methoden `compare` und `equals` stehen:

`Object`-Methoden ausgenommen

```
interface Comparator<T> {
    int compare(T first, T second);
    boolean equals(Object x);
}
```

[23] Leere Interfaces dienen zur Markierung gewisser Eigenschaften und werden folglich als „Marker-Interfaces“ bezeichnet. Ein Beispiel ist `java.io.Serializable`. Die Frage, ob Interfaces für diesen Zweck überhaupt das angemessene Ausdrucksmittel sind, ist nicht ganz unberechtigt. Eine Alternative bieten Marker-Annotationen, siehe Seite 586.

[24] Abstrakte Basisklassen sind keine Interfaces und scheiden daher aus, auch wenn sie genau eine abstrakte Methode definieren.

Eine Implementierung muss von diesen beiden aber nur `compare` explizit definieren und kann die ererbte Fasung von `equals` übernehmen.[25]

Auch generische Interfaces, die nur eine Methode definieren, sind Funktionsinterfaces, wie die folgenden Beispiele zeigen:

```
public interface Block<T> {
    void apply(T x);
}
```

Listing 5.23: Ein Codeblock mit einem Parameter, der nichts zurückliefert und durch Seiteneffekte wirkt.

```
public interface Mapper<T, U> {
    U map(T t);
}
```

Listing 5.24: Eine Abbildung eines Argumentes vom Typ `T` auf ein Ergebnis vom Typ `U`.

```
public interface Operator<T> {
    T eval(T x, T y);
}
```

Listing 5.25: Eine Operation, die zwei Operanden zu einem Ergebnis verknüpft.

Im letzten Beispiel ist ein Funktionsinterface von einem anderen abgeleitet:

```
interface Function<T> extends Mapper<T, T> {
}
```

Listing 5.26: Eine Funktion, die ein Argument auf ein Ergebnis vom gleichen Typ abbildet.

5.5.2 Lambda-Ausdrücke

Literal eines Objekts kompatibel zu Funktionsinterface

Ein **Lambda-Ausdruck** ist das Literal eines Objekts, das ein Funktionsinterface implementiert. Ein solches Objekt wird im Folgenden als **Funktionsobjekt** bezeichnet. Syntaktisch besteht ein Lambda-Ausdruck aus

[25] `equals` ist dennoch ausdrücklich im Interface genannt, weil die von `Object` ererbte Fassung zwar den Compiler zufriedenstellt, aber trotzdem in der Regel nicht ausreicht. `compare` und `equals` *müssen* bezüglich Gleichheit im Einklang stehen.

- einer Parameterliste und
- einem Methodenrumpf,

die mit dem Operator `->` verknüpft sind:

```
(params) -> {...}
```

Dieser Ausdruck ist zuweisungskompatibel zu jedem Funktionsinterface, dessen einzige Methode die passende Parameterliste und den passenden Ergebnistyp verlangt.

Lambda-Ausdruck zum Funktionsinterface `Runnable`

Das Interface `Runnable` definiert zum Beispiel die Methode `run` ohne Parameter mit Rückgabetyp `void`:

```
interface Runnable {
   void run();
}
```

Der Lambda-Ausdruck

```
() -> {System.out.println("Hello, Lambda!");}
```

ist kompatibel zu `Runnable`, weil er auch eine leere Parameterliste hat und kein Ergebnis liefert. Die folgende Wertzuweisung weist den Lambda-Ausdruck an eine `Runnable`-Variable zu:

```
Runnable r = () -> {System.out.println("Hello, Lambda!");};
```

Mit dem `Runnable`-Objekt kann wie gewohnt verfahren werden. Es kann dem `Thread`-Konstruktor übergeben und der Thread dann gestartet werden. Die Ausgabe des Lambda-Ausdrucks läuft parallel ab, auch wenn das in diesem Beispiel äußerlich nicht zu erkennen ist:

```
Thread th = new Thread(r);
th.start();        // gibt "Hello, Lambda!" aus
```

Lambda-Ausdruck als Argument

Die rechte Seite des Zuweisungsoperators ist nicht der einzige Kontext, in dem ein Lambda-Ausdruck stehen kann. Auch die Argumentliste eines Methodenaufrufs verbirgt (implizite) Wertzuweisungen der Argumente an die Parameter. Im folgende Beispiel dient der Lambda-Ausdruck als Argument des `Thread`-Konstruktors:

```
Thread th = new Thread(() -> {System.out.println("Hello, Lambda!");});
th.start();
```

Das nächste Programm startet zwei Threads, die jeweils einen Gruß ausgeben. `main` schiebt noch eine dritte Ausgabe nach.

```
public class RunnableLambda {
    public static void main(String... args) {
        new Thread(() -> {System.out.println("Hello");}).start();
        new Thread(() -> {System.out.println("Hello, again");}).start();
        System.out.println("Hello, once again");
    }
}
```

Listing 5.27: Lambda-Ausdruck als Implementierung der `run`-Methode des Interface `Runnable`.

Auf dem Bildschirm erscheinen alle drei Ausgaben:[26]

```
$ java RunnableLambda
Hello
Hello, once again
Hello, again
```

Lambda-Ausdruck als Rückgabewert

Auch bei der Ergebnisrückgabe einer Methode läuft eine implizite Wertzuweisung ab. Im folgenden Programm liefert die Methode `makeRunnable` ein `Runnable`-Objekt zurück, das aus einem Lambda-Ausdruck in der `return`-Anweisung stammt:

```
public class LambdaResult {
    static Runnable makeRunnable() {
        return () -> {System.out.println("Hello");};
    }

    public static void main(String... args) {
        new Thread(makeRunnable()).start();
        System.out.println("Hello, once again");
    }
}
```

Listing 5.28: Lambda-Ausdruck als Rückgabewert einer Methode.

Syntaktische Abkürzungen

Bei der Formulierung von Lambda-Ausdrücken sind abkürzende Schreibweisen erlaubt. Diese Schreibweisen eröffnen keine neuen Möglichkeiten, sondern führen nur unter bestimmten Voraussetzungen zu leichter lesbarem Quelltext.

[26] Die Reihenfolge der Ausgaben ist nicht vorhersagbar, weil die JVM die Threads nach eigenen Maßgaben startet.

1. Der Rumpf eines Lambda-Ausdrucks, der nur aus einer einzigen `return`-Anweisung mit einem Ergebniswert besteht, kann durch den Ausdruck alleine ersetzt werden:

   Lambda-Ausdruck mit einer `return`-Anweisung als Rumpf

   ```
   (params) -> {return expression;}
   (params) -> expression
   ```

   Beispielsweise kann der `Comparator` in `CustomStringOrdering` (Listing 5.18) durch die beiden folgenden äquivalenten Lambda-Ausdrücke ersetzt werden:

   ```
   (String s0, String s1) -> {return s0.length() - s1.length();}
   (String s0, String s1) -> s0.length() - s1.length()
   ```

   Das folgende Programm zeigt eine Anwendung:

   ```
   import java.util.*;

   public class StringLengthLambda {
       public static void main(String... args) {
           Comparator<String> comparator =
               (String s0, String s1) -> s0.length() - s1.length();
           TreeSet<String> set = new TreeSet<>(comparator);
           for(String arg: args)
               set.add(arg);
           System.out.println(set);
       }
   }
   ```

   Listing 5.29: Lambda-Ausdruck, der nur einen Ausdruck zurückgibt.

   Die Variable `comparator` ist dabei nicht unbedingt nötig. Der Lambda-Ausdruck kann auch direkt als Argument an den `TreeSet`-Konstruktor übergeben werden, um den Code weiter zu vereinfachen.

2. Die *Type-Inference* des Compilers kann oft die Parametertypen eines Lambda-Ausdrucks aus dem Zuweisungskontext erschließen. In diesen Fällen können die Typen der Parameter weggelassen werden.[27] Der Rumpf des Lambda-Ausdrucks fließt dagegen *nicht* in die Type-Inference ein.

   Type-Inference der Parametertypen

   Wenn der Compiler ausreichend Typinformationen findet, sind beide folgenden Lambda-Ausdrücke gleichwertig:

   ```
   (String s0, String s1) -> s0.length() - s1.length();
   (s0, s1) -> s0.length() - s1.length();
   ```

   Im folgenden Beispiel ist die Type-Inference erfolgreich, deshalb können die Parametertypen wegbleiben:

   ```
   import java.util.*;
   ```

[27] Allerdings muss man sich entscheiden: Entweder werden alle Parametertypen angegeben oder alle weggelassen. Mischungen sind nicht erlaubt.

```
public class StringLengthLambda1 {
    public static void main(String... args) {
        Comparator<String> comparator = (s0, s1) -> s0.length() - s1.length();
        TreeSet<String> set = new TreeSet<>(comparator);
        for(String arg: args)
            set.add(arg);
        System.out.println(set);
    }
}
```

Listing 5.30: Lambda-Ausdruck mit automatisch erschlossenen Parametertypen.

Verschiebt man den Lambda-Ausdruck als Argument in den `TreeSet`-Konstruktor, dann verliert der Compiler den Faden:

```
TreeSet<String> set = new TreeSet<>((s0, s1) -> s0.length() - s1.length());
```

Er meldet einen Fehler:[28]

```
error: cannot infer type arguments for TreeSet<>
```

Lambda-Ausdruck mit einem Parameter

3. Ein einziger Parameter ohne Typangabe kann ohne runde Klammern geschrieben werden:[29]

```
(name) -> ...
name -> ...
```

Beispielsweise ist das Interface `PathProcessor` (Listing 4.28) ein Funktionsinterface. Die Anwendung im Programm `PrintFiletree` (Listing 4.30) kann mit einem Lambda-Ausdruck mit einem Parameter formuliert werden, dessen Typ (`Path`) der Compiler ergänzt:

```
import java.io.*;
import java.nio.file.*;

public class PrintFiletreeLambda {
    public static void main(String... args) throws IOException {
        new DirWalker(path -> {System.out.println(path);},
                      null,
                      null,
                      null).walk(Paths.get(args[0]));
    }
}
```

Listing 5.31: Ein Lambda-Ausdruck als Visitor.

Auch das Interface `PermutationEater` (Listing 4.40) ist ein Funktionsinterface. Mit einem Lambda-Ausdruck fällt die Anwendung `PermutationPrinter` (Listing 4.42) einfacher aus:

[28] Verzichtet man auf den Diamond-Operator im `TreeSet`-Konstruktor und gibt das Typargument `String` ausdrücklich an, dann schafft die Type-Inference des Compilers wieder den Sprung zum Lambda-Ausdruck.

[29] Das gilt nicht für eine leere Parameterliste, die immer als `()` angegeben werden muss.

```
import java.util.*;

public class PermutationPrinterLambda {
    public static void main(String... args) {
        Set<String> elements = new HashSet<>(Arrays.asList(args));
        PermutationGenerator.generate(new ArrayList<String>(),
                                      elements,
                                      p -> {System.out.println(p);});
    }
}
```

Listing 5.32: Ein Lambda-Ausdruck als Konsument von generierten Permutationen.

5.5.3 Lambda-Ausdrücke im Typsystem

Lambda-Ausdrücke sind eine neue syntaktische Struktur in Java. Sie sind keine normalen Referenztypen. Beispielsweise wird die Zuweisung

Lambda-Ausdrücke als neuer Typ

```
Object lambda = x -> x + 1;    // Fehler
```

nicht übersetzt. Selbst der universelle Gleichheitsoperator == ist nicht anwendbar:

```
System.out.println((x -> x + 1) == (x -> x + 1));    // Fehler
```

Ein isolierter Lambda-Ausdruck hat *keinen bestimmten* Typ.[30] Er passt sich erst bei der Zuweisung an einen **Zieltyp** an, der ein Funktionsinterface sein muss. Ab diesem Zeitpunkt bestimmt der Zieltyp die Arbeitsweise des Lambda-Ausdrucks.

Anpassung an einen Zieltyp

Im folgenden Beispiel steht der Lambda-Ausdruck

Kontext bestimmt Arbeitsweise

```
(s0, s1) -> s0 + s1
```

in verschiedenen Kontexten mit unterschiedlichen Zieltypen. Die Methode `fold` wendet das Funktionsobjekt `op` nacheinander auf die Elemente `elements` an. Das Ergebnis der vorhergehenden Verknüpfung liefert jeweils den Operanden der nächsten Verknüpfung. Den Anfang macht der Parameter `starter`:

```
public class ArrayFolder {
    static <T> T fold(Operator<T> op, T starter, T... elements) {
        T result = starter;
        for(T element: elements)
```

[30] Einem Lambda-Ausdruck wird ein *poly type* zugesprochen, der die Menge aller passenden Funktionsinterfaces umfasst.

```
            result = op.eval(result, element);
        return result;
    }

    public static void main(String... args) {
        System.out.println(fold((s0, s1) -> s0 + s1, "", "1", "2", "3"));
        System.out.println(fold((s0, s1) -> s0 + s1, 0, 1, 2, 3));
    }
}
```

Listing 5.33: Type-Inference im lokalen Kontext.

Das Programm gibt aus:

```
$ java ArrayFolder
123
6
```

Im Einzelnen kann eine Lambda-Ausdruck in den folgenden Situationen verwendet werden:

1. Rechte Seite einer Wertzuweisung oder Variablen-Initialisierung
2. Werteliste eines Array-Literals
3. Argumentposition
4. Ergebnisrückgabe mit `return`
5. Rumpf eines größeren Lambda-Ausdrucks
6. Operand eines bedingten Ausdrucks (dreistelliger Operator `?:`)
7. Operand eines Typecast auf ein Funktionsinterface

Diese Liste begrenzt die Einsatzmöglichkeiten von Lambda-Ausdrücken. Nur dort, und nirgendwo sonst, darf ein Lambda-Ausdruck genannt werden.

Zuweisung an unterschiedliche Funktionsinterfaces

Ein Lambda-Ausdruck besteht nur aus Parametern und Rumpf, legt aber keinen Methodennamen fest. Diesen steuert erst der Zieltyp bei. Im folgenden Beispiel wird der gleiche Lambda-Ausdruck an Variablen zweier verschiedener Funktionsinterfaces `Foo` und `Bar` zugewiesen, die nichts miteinander zu tun haben und insbesondere auch nicht zueinander kompatibel sind:

```
interface Foo {
    int foo();
}

interface Bar {
    int bar();
```

```
}

public class LambdaMethodName {
    public static void main(String... args) {
        Foo f = () -> 42;
        Bar b = () -> 42;
        System.out.println(f.foo());
        System.out.println(b.bar());
    }
}
```

Listing 5.34: Aufruf eines Lambda-Ausdrucks mit verschiedenen Methodennamen.

Der Lambda-Rumpf wird mit unterschiedlichen Methodennamen aufgerufen.

5.5.4 Closure

Einschluss lokaler Variablen der Definitionsumgebung

Effektiv unveränderliche Variablen

Der Rumpf eines Lambda-Ausdrucks hat freien Zugriff auf alle privaten Variablen in der umgebenden Toplevel-Klasse und auf die lokalen Variablen am Ort der Definition. Er schließt diese Variablen in einer Closure (siehe Seite 319) ein und nimmt sie mit sich, sodass sie beim Aufruf der Closure in einer möglicherweise ganz anderen Umgebung wieder zur Verfügung stehen. Dabei sind allerdings nur **effektiv unveränderliche** lokale Variablen erlaubt. Eine Variable ist „effektiv unveränderlich“, wenn sie mit dem Modifier `final` *versehen werden könnte* und das Programm dann immer noch übersetzt werden würde.[31]

Das folgende Beispiel illustriert dieses Verhalten. `main` definiert eine Liste `list` mit Objekten vom Typ `Code` (Listing 5.22). Die erste Schleife füllt die Liste mit Lambda-Ausdrücken, von denen jeder ein Kommandozeilenargument ausgibt. Eine zweite Schleife führt die `Code`-Objekte nacheinander aus.

```
import java.util.*;

public class LambdaList {
    public static void main(String... args) {
        List<Code> list = new ArrayList<>();
        for(String arg: args)
            list.add(() -> {System.out.println(arg);});
        for(Code code: list)
            code.run();
    }
}
```

Listing 5.35: Erzeugen von Lambda-Ausdrücken, die die Schleifenvariable referenzieren, in einer *foreach*-Schleife.

[31] Das bedeutet, dass technisch die gleichen Einschränkungen wie bei lokalen und anonymen Klassen in bisherigen Java-Versionen gelten. Allerdings muss `final` in Java 8 nicht mehr explizit angegeben werden.

Viele Variablen effektiv unveränderlich

Die Schleifenvariable `String arg` ist effektiv unveränderlich, obwohl der Modifier `final` nicht genannt ist, und steht deshalb dem Lambda-Ausdruck zur Verfügung. Das Gleiche gilt für alle anderen Variablen im Programm, das heißt für den `main`-Parameter `args`, die Liste `list` und die Schleifenvariable `code`. Sie alle könnten mit `final` versehen werden, ohne das Programm zu ändern.

```
$ java LambdaList abra ka dabra
abra
ka
dabra
```

Formuliert man das Programm dagegen mit einer Indexschleife, die das Array der Kommandozeilenargumente durchläuft, dann wird es nicht mehr übersetzt. Die Indexvariable `arg` ist hier *nicht* effektiv unveränderlich und ist für den Lambda-Ausdruck tabu:

```
import java.util.*;

public class LambdaMutableBroken {
    public static void main(String... args) {
        List<Code> list = new ArrayList<>();
        for(int arg = 0; arg < args.length; arg++)
            list.add(() -> {System.out.println(args[arg]);}); // Fehler: arg nicht final
        for(Code code: list)
            code.run();
    }
}
```

Listing 5.36: Erzeugen von Lambda-Ausdrücken, die den veränderlichen Zähler einer `for`-Schleife referenzieren.

5.5.5 Gültigkeitsbereiche

Definitionsumgebung von Lambda-Ausdrücken

Ein Lambda-Ausdruck definiert keinen eigenen Gültigkeitsbereich. Er ist Teil der Definitionsumgebung und übernimmt deren Variablen.

Infolgedessen kollidieren Parameter und lokale Variablen in einem Lambda-Ausdruck mit lokalen Variablen der Definitionsumgebung. Anonyme Klassen verhalten sich in diesem Punkt anders als Lambda-Ausdrücke, weil sie einen eigenen Gültigkeitsbereich abgrenzen.

Bedeutung von `this` und `super`

`this` in Lambda-Ausdrücken

Aus der obigen Regelung ergibt sich die unterschiedliche Rolle von `this` in Lambda-

Ausdrücken und in anonymen Klassen. In einem Lambda-Ausdruck bezieht sich `this` gemäß lexikalischer Bindung auf die Definitionsumgebung. In einer anonymen Klasse referenziert `this` das Objekt der namenlosen Klasse.

Das folgende Programm macht diesen Unterschied sichtbar, denn es wird *nicht* übersetzt. Der Aufruf von `toString` richtet sich an `this`, das aber in der statischen Methode `main` nicht definiert ist:

```
public class LambdaThisBroken {
    public static void main(String... args) {
        Code b = () -> {System.out.println(toString());};    // wird nicht übersetzt
        b.run();
    }
}
```

Listing 5.37: Bezug von `this` in einem Lambda-Ausdruck auf die Definitionsumgebung.

`this` in einer anonymen Klasse

Ersetzt man den Lambda-Ausdruck auf der rechten Seite der Wertzuweisung durch eine anonyme Klasse, dann existiert `this`. Der Code wird übersetzt und läuft:

```
public class AnonymousThis {
    public static void main(String... args) {
        Code b = new Code() {
            public void run() {
                System.out.println(toString());
            }
        };
        b.run();
    }
}
```

Listing 5.38: `this` in einer anonymen Klasse.

Entsprechendes gilt für `super`, die Referenz auf das Basisklassenobjekt.

`break` und `continue`

Kein `break` und `continue` aus Lambda-Ausdruck heraus

Die Anweisungen `break` und `continue` können nicht aus einem Lambda-Ausdruck heraus in eine umgebende Kontrollstruktur springen. Das folgende Programm wird nicht übersetzt:

```
public class BreakLambdaBroken {
    public static void main(String... args) {
        Block<String> empty = s -> {if(s.isEmpty()) break;}; // wird nicht übersetzt
```

```
    for(String arg: args) {
            empty.apply(arg);
            System.out.println(arg);
        }
    }
}
```

Listing 5.39: Keine Sprünge aus Lambda-Ausdrücken heraus.

`break` und `continue` können durchaus in einem Lambda-Ausdruck vorkommen, dort aber nur als Teil einer Kontrollstruktur, die komplett im Lambda-Ausdruck enthalten ist.

5.5.6 Exceptions in einem Lambda-Ausdruck

Der Rumpf eines Lambda-Ausdrucks kann Exceptions auslösen. Checked-Exceptions muss die Methodensignatur im Funktionsinterface ankündigen, während Unchecked-Exceptions nicht erwähnt werden müssen.

Das folgende Beispielprogramm konstruiert eine Liste von Funktionsobjekten des Typs `Code`, von denen jedes ein Kommandozeilenargument nach drei Zeichen abschneidet.

```
import java.util.*;

public class LambdaThrow {
    public static void main(String... args) {
        List<Code> list = new ArrayList<>();
        for(String arg: args)
            list.add(() -> {System.out.println(arg.substring(3));});
        for(Code code: list)
            try {
                code.run();
            }
            catch(Exception ex) {
                System.out.println("caught " + ex);
            }
    }
}
```

Listing 5.40: Fangen einer Exception in der Ablaufumgebung.

Zu kurze Kommandozeilenargumente können nicht abgeschnitten werden und lösen daher eine Exception aus. Diese Exception tritt beim Aufruf des Funktionsobjekts, das heißt in der zweiten Schleife auf.

```
$ java LambdaThrow Abra ka dabra
a
caught StringIndexOutOfBoundsException: index out of range: -1
ra
```

5.5.7 Geschachtelte Lambda-Ausdrücke

Lambda-Ausdruck als Ergebnis eines anderen Lambda-Ausdrucks

Der Rumpf eines Lambda-Ausdrucks ist ein Kontext, in dem weitere Lambda-Ausdrücke verwendet werden können. Das bedeutet, dass ein Lambda-Ausdruck einen anderen als Ergebnis liefern kann.

Lambda-Ausdruck als Ergebnis eines Lambda-Ausdrucks

Als Beispiel dient ein „äußerer“ Lambda-Ausdruck, der eine Zahl `x` als Argument erwartet und einen „inneren“ Lambda-Ausdruck produziert. Dieser innere Lambda-Ausdruck erwartet selbst eine weitere Zahl `y` und liefert die Summe `x + y`:

```
(Integer y) -> {return x + y;}
```

Dieser Ausdruck ist der Rückgabewert des äußeren Lambda-Ausdrucks mit dem Parameter `x`:

```
(Integer x) -> {return (Integer y) -> {return x + y;};}
```

Eine einfachere Schreibweise dieses Ausdrucks ist:

```
x -> y -> x + y
```

Die Parameter- und Ergebnistypen der Lambda-Ausdrücke legen die Typen passender Funktionsinterfaces fest. Der innere Lambda-Ausdruck ist kompatibel zu

```
Function<Integer>
```

Der äußere Lambda-Ausdruck bildet eine Zahl auf diesen Typ ab und ist daher kompatibel zu

```
Mapper<Integer, Function<Integer>>
```

Das folgende Programm gibt 6, 11, 4 und 9 aus:

```
import static java.lang.System.*;
import java.util.functions.*;

public class MakeAdder {
    public static void main(String... args) {
        Mapper<Integer, Function<Integer>> makeAdder = x -> y -> x + y;

        Function<Integer> inc = makeAdder.map(1);
        out.println(inc.map(5));
        out.println(inc.map(10));

        Function<Integer> dec = makeAdder.map(-1);
        out.println(dec.map(5));
        out.println(dec.map(10));
    }
}
```

Listing 5.41: Lambda-Ausdruck als Rückgabewert eines größeren Lambda-Ausdrucks.

Lambda-Ausdruck als Argument eines Lambda-Ausdrucks

Funktionsinterface als Lambda-Parameter

Auch das Argument eines Lambda-Ausdrucks kann ein anderer Lambda-Ausdruck sein. Der Lambda-Parameter hat in diesem Fall den Typ eines Funktionsinterface. Im folgenden Beispiel erwartet ein Lambda-Ausdruck einen `Block` (Listing 5.23) `b` als Argument: Er liefert als Ergebnis einen inneren Lambda-Ausdruck mit einem Parameter `t`, der `b` zweimal nacheinander aufruft. Der innere Lambda-Ausdruck lautet:

```
(T t) -> {b.apply(t); b.apply(t);}
```

Dieser Ausdruck ist der Rückgabewert des äußeren Lambda-Ausdrucks:

```
(Block<T> b) -> {return (T t) -> {b.apply(t); b.apply(t);};}
```

oder kürzer:

```
b -> t -> {b.apply(t); b.apply(t);}
```

Der Parameter und das Ergebnis des äußeren Lambda-Ausdrucks haben den gleichen Typ `Block<T>`. Der ganze Ausdruck ist also kompatibel zu

```
Function<Block<T>>
```

Lambda-Ausdruck zur mehrfachen Anwendung eines Blocks

Das folgende Programm definiert ein Objekt `twice`, das ein Funktionsobjekt vom Typ `Block` für Strings zweimal nacheinander ausführt. Mit `twice` werden neue Blöcke berechnet: `print2` gibt sein Argument zweimal aus, `adder2` fügt es zweimal an eine Liste an und `adder8` besteht aus drei verschachtelten Blöcken:

```
import java.util.*;
import java.util.functions.*;

public class TwiceLambda {
    public static void main(String... args) {
        Function<Block<String>> twice =
            b -> t -> {b.apply(t); b.apply(t);};

        Block<String> print2 = twice.map(s -> {System.out.println(s);});
        print2.apply(args[0]);

        final List<String> list = new ArrayList<>();
        Block<String> adder2 = twice.map(s -> {list.add(s);});
        adder2.apply(args[1]);
        System.out.println(list);

        Block<String> adder8 = twice.map(twice.map(adder2));
        adder8.apply(args[2]);
        System.out.println(list);
    }
}
```

Listing 5.42: Lambda-Ausdruck, der einen anderen als Argument akzeptiert und ein Funktionsobjekt liefert.

Das Programm druckt das erste Kommandozeilenargument zweimal aus, dann die Liste mit zwei Elementen und noch einmal die Liste mit acht weiteren Elementen:

```
$ java TwiceLambda sim sala bim
sim
sim
[sala, sala]
[sala, sala, bim, bim, bim, bim, bim, bim, bim, bim]
```

5.5.8 Funktionsinterfaces in der Bibliothek

Package mit allgemeinen Funktionsinterfaces

Die oben eingeführten einfachen Funktionsinterfaces `Block` (Listing 5.23), `Mapper` (Listing 5.24) und `Operator` (Listing 5.25) werden oft gebraucht. Deshalb sind sie im Package `java.util.functions` vordefiniert. Dort finden sich insgesamt die folgenden Funktionsinterfaces:

`interface Operator<T> {T eval(T left, T right);}`
: Kombiniert zwei Operanden zu einem Ergebnis. Die Operanden bleiben unverändert.

`interface Predicate<T> {boolean eval(T t);}`
: Stellt eine Eigenschaften des Operanden fest. Der Operand bleibt unverändert.

`interface Mapper<T, U> {U map(T t);}`
: Bildet den Operanden `t` auf einen anderen Wert ab. Der Operand bleibt unverändert.

`interface Block<T> {void apply(T t);}`
: Wendet eine Operation auf den Operanden `t` an, der dabei verändert werden kann.

Die Aussagen zur Veränderung der Operanden sind nicht technisch abgesichert, sondern *sollten* eingehalten werden. Ein `Block`-Objekt darf beispielsweise den Operanden verändern, um Informationen an den Aufrufer zurückzuliefern. Objekte der anderen Typen sollen das nicht tun.

5.6 Default-Methoden (Java 8)

Bibliotheksklassen erweitert für Funktionsobjekte

Java 8 erweitert viele Bibliotheksklassen und -Interfaces um Methoden, die mit Funktionsobjekten arbeiten. Beispielsweise erhält das Interface `Iterable` eine neue Methode:

```
Iterable<E>  filter(Predicate<? super E> predicate)
```

`filter` liefert ein neues `Iterable` mit den Elementen, die die Eigenschaften `predicate` haben.[32] Das folgende Programm sucht aus der Liste mit den Kommandozeilenargumenten alle Strings heraus, die kürzer als drei Zeichen sind:

```
import java.util.*;

public class StripShortStrings {
    public static void main(String... args) {
        List<String> arglist = Arrays.asList(args);
        for(String arg: arglist.filter(s -> s.length() < 3))
            System.out.println(arg);
    }
}
```

Listing 5.43: Anwendung eines Funktionsobjektes auf alle Elemente einer Liste.

Beispiel: Interface `Iterable` in Java 8

Bisher (Java 7) definiert das Interface `Iterable<T>` nur eine einzige Methode:

```
Iterator<T>  iterator();
```

[32] Genauer gesagt liefert `filter` eine *Sicht* auf das ursprüngliche `Iterable`, die nur die ausgewählten Elemente zeigt und die übrigen ausblendet. Das heißt, dass sich Änderungen am Original auf die Sicht auswirken. Eine Sicht ist keine unabhängige Datenstruktur.

Außer der oben gezeigten Methode `filter` kommen in der Bibliothek von Java 8 viele weitere hinzu:[33]

`Iterable<T> filter(Predicate<T> predicate)`
: Liefert ein neues `Iterable` mit nur den Elementen, die das Prädikat erfüllen.

`Iterable<T> forEach(Block<T> block)`
: Führt den Block mit jedem Element aus und liefert ein neues `Iterable` mit den möglicherweise veränderten Elementen.

`<U> Iterable<U> map(Mapper<T, U> mapper)`
: Liefert ein neues `Iterable` mit den Ergebnissen der Anwendung des Mappers auf jedes Element.

`T reduce(T base, Operator<T> reducer)`
: Wendet den Operator nacheinander auf jedes Element und das Ergebnis der vorhergehenden Anwendung an. Den Start macht das erste Element mit `base`.

`<U> U mapReduce(Mapper<T, U> mapper, U base, Operator<U> reducer)`
: Wie `reduce`, bildet aber die Elemente vorher mit dem Mapper ab.

`boolean anyMatch(Predicate<T> filter)`
: Stellt fest, ob irgendein Element das Prädikat erfüllt.

`boolean noneMatch(Predicate<T> filter)`
: Stellt fest, ob kein Element das Prädikat erfüllt.

`boolean allMatch(Predicate<T> filter)`
: Stellt fest, ob jedes Element das Prädikat erfüllt.

Ungültigkeit bestehender Implementierungen

Mit solchen Erweiterungen *bestehender Interfaces* wären allerdings alle bereits existierenden Implementierung der Interfaces mit einem Schlag ungültig, weil sie die neuen Methoden nicht implementieren und damit unvollständig wären.

Default-Methoden erlauben nachträgliche Erweiterung

Das Problem lösen **Default-Methoden** in Interfaces. Sie stehen in keinem direkten Zusammenhang zu Lambda-Ausdrücken und geschachtelten Typdefinitionen, dem Hauptthema dieses Kapitels. Dennoch werden Default-Methoden an dieser Stelle diskutiert, weil die Lambda-Ausdrücke der wesentliche Auslöser für ihre Einführung sind.

[33] Das Katalog ist noch länger. Die Wildcard-Typen in den Methodensignaturen sind zur besseren Lesbarkeit vereinfacht.

5.6.1 Definition von Default-Methoden

Methoden mit Rümpfen in Interfaces

In Java 8 können Interfaces neben abstrakten Methoden auch Default-Methoden enthalten. Syntaktisch unterscheiden sich Default-Methoden von abstrakten Methoden durch einen Rumpf, der nach dem Schlüsselwort `default`[34] angefügt wird:

```
methodhead default body
```

Der Methodenrumpf unterliegt gewissen Einschränkungen, folgt aber ansonsten der üblichen Syntax.

Implementierungen *können* Default-Methoden redefinieren

Implementierungen *müssen* wie bisher alle abstrakten Methoden definieren. Sie *können* darüber hinaus die Default-Methoden redefinieren, müssen das aber nicht tun. Eine Klasse, die eine Default-Methoden nicht selbst neu definiert, arbeitet automatisch mit der Implementierung im Interface.

Als Beispiel dient das folgende Interface für Rechtecke. Es verlangt Getter für die Breite und die Höhe des Rechtecks. Darüber hinaus definiert es die Default-Methode `isSquare`, die Auskunft gibt, ob das Rechteck quadratisch ist oder nicht:

```
public interface Rectangle {
    double getWidth();

    double getHeight();

    boolean isSquare() default {
        return getWidth() == getHeight();
    }

}
```

Listing 5.44: Interface mit einer Default-Methode.

5.6.2 Implementierung von Default-Methoden

Konkrete Klassen können Default-Methoden eines Interface redefinieren oder die vorgegebene Implementierung übernehmen.

Die folgende Klasse implementiert das Interface `Rectangle` (Listing 5.44). Sie definiert die abstrakten Methoden `getWidth` und `getHeight`, nicht aber `isSquare`:

[34] Das Schlüsselwort `default` dient auch als Label in `switch`-Anweisungen. Der syntaktische Kontext ist in den beiden Fällen allerdings völlig verschieden, sodass keine Verwechslungsgefahr besteht.

```
public class SimpleRectangle implements Rectangle {
    private final double width;

    private final double height;

    public SimpleRectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double getHeight() {
        return height;
    }

    public double getWidth() {
        return width;
    }
}
```

Listing 5.45: Implementierung eines Interface ohne Redefinition der Default-Methode.

Eine andere Klasse repräsentiert nur Quadrate. Sie implementiert ebenfalls das Interface `Rectangle`, redefiniert aber `isSquare` mit einer einfacheren Implementierung:

```
public class Square implements Rectangle {
    private final double size;

    public Square(double size) {
        this.size = size;
    }

    public double getHeight() {
        return size;
    }

    public double getWidth() {
        return size;
    }

    public boolean isSquare() { // Redefinition der Default-Methode
        return true;
    }
}
```

Listing 5.46: Implementierung eines Interface mit expliziter Redefinition einer Default-Methode.

Default-Methode für Aufrufer nicht erkennbar

Für eine Anwendung spielt es keine Rolle, ob ein Objekt eine Default-Methode oder eine eigene Implementierung benutzt:

```
public class RectangleMain {
```

```
    public static void main(String... args) {
        Rectangle[] rectangles = new Rectangle[] {
            new SimpleRectangle(1, 2),
            new SimpleRectangle(1, 1),
            new Square(2),
        };
        for(Rectangle rectangle: rectangles)
            System.out.println(rectangle.isSquare());
    }
}
```

Listing 5.47: Programm mit Aufrufen von Default-Methoden.

Das Programm produziert die erwarteten Ergebnisse. Die ersten beiden Ausgaben stammen von Default-Methoden, die dritte von einer expliziten Implementierung:

```
$ java RectangleMain
false
true
true
```

5.6.3 Einschränkungen von Default-Methoden

Keine Objektvariablen für Default-Methoden

Default-Methoden sind in Interfaces definiert. Im Gegensatz zur bisherigen Leitlinie enthalten Interfaces damit erstmals Code.

Default-Methoden unterliegen einer wichtigen Einschränkung, die ihre Existenz in Interfaces legitimiert: Ihnen stehen keinerlei Objektvariablen zur Verfügung.[35] Default-Methoden können sich gegenseitig und auch die abstrakten Methoden des eigenen Interface aufrufen. Bei abgeleiteten Interfaces kommen noch die abstrakten Methoden und Default-Methoden aller direkten und indirekten Basis-Interfaces hinzu. Von allen diesen Methoden kann aber keine Objektvariablen ansprechen.

Das bedeutet, dass implementierende Klassen mit Default-Methoden zwar *Operationen*, aber keinen *Zustand* hinzugewinnen. Alle veränderlichen Informationen eines Objekts sind also weiterhin ausschließlich in Klassen definiert.

5.6.4 Vererbung und Default-Methoden

Verschiedene Mechanismen mit gleichem Ergebnis

Vererbung macht Methoden in Klassen verfügbar, die dort nicht explizit definiert

[35] Interfaces erlauben zwar Variablendefinitionen, der Compiler versieht diese aber automatisch mit `public`, `static` und `final`. Es handelt sich also nur um öffentliche Konstanten.

sind. Default-Methoden bewirken im Prinzip das Gleiche und konkurrieren in dieser Hinsicht mit Vererbung.

Ein Konflikt ergibt sich beim Aufruf einer Methode, die sowohl ererbt wird, als auch als Default-Methode definiert ist. In diesem Fall gilt eine einfache Regel:

Ererbte Methode vor Default-Methode.

Betrachten Sie zum Beispiel die folgende Klasse mit nur einer Methode `isSquare`:

```
public class Polygon {
    public boolean isSquare() {
        return false;
    }
}
```

Listing 5.48: Basisklasse für Polygone.

Die Klasse `DerivedRectangle` erbt von `Polygon` *und* implementiert das Interface `Rectangle`. Sowohl die Basisklasse als auch das Interface stellen die Methode `isSquare` bereit, die `DerivedRectangle` selbst nicht definiert:

Ererbte Methoden vor Default-Methoden

```
public class DerivedRectangle extends Polygon implements Rectangle {
    // ... wie SimpleRectangle, also ohne isSquare

}
```

Listing 5.49: Abgeleitete Klasse erbt eine Methode und erhält eine Default-Methode.

Gemäß der obigen Regel richtet sich ein Aufruf von `isSquare` an die ererbte Fassung und nicht an die Default-Implementierung. Das folgende Programm gibt deshalb `false` aus:

```
public static void main(String[] args) {
    DerivedRectangle rectangle = new DerivedRectangle(2, 2);
    System.out.println(rectangle.isSquare());
}
```

Listing 5.50: Ererbte Methode verdrängt Default-Methode.

Konkurrierende Default-Methoden

Eine Klasse kann nur von *einer* Basisklasse erben, aber sie kann mehrere Interfaces

Default-Implementierungen in verschiedenen Interfaces

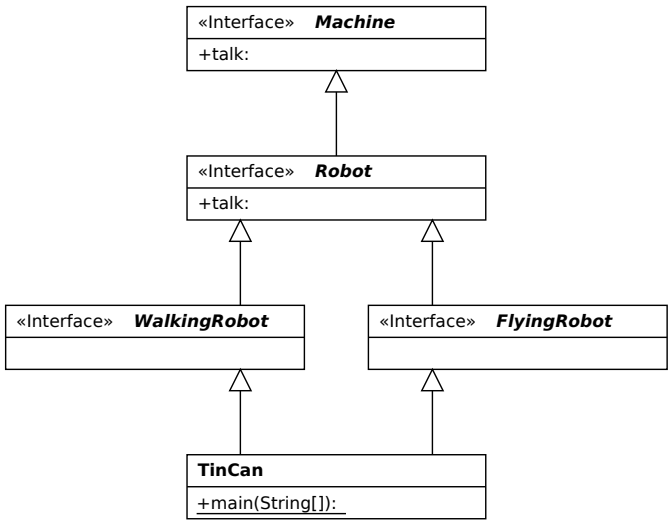
implementieren. Einer Klasse können damit unterschiedliche Implementierungen *der gleichen* Default-Methode zur Verfügung stehen. Diese Situation ähnelt auf den ersten Blick der Mehrfachvererbung, die es in Java aus guten Gründen nicht gibt.

Mehrfache Default-Methoden

Der Compiler sammelt beim Übersetzen einer Klasse zunächst alle infrage kommenden Default-Methoden in allen direkt oder indirekt implementierten Interfaces ein. Aus dieser Menge streicht er Kandidaten, die von anderen Kandidaten redefiniert werden. Es bleiben also nur die am weitesten abgeleiteten Default-Methoden erhalten. Am Ende muss *genau ein* Kandidat übrig bleiben. Andernfalls bricht der Compiler mit der gezeigten Fehlermeldung ab.

Die folgende Typhierarchie illustriert dieses Vorgehen:

- Das Interface `Machine` repräsentiert eine beliebige Maschine.
- Abgeleitet von `Machine` ist das Interface `Robot` für autonome Systeme.
- `WalkingRobot` und `FlyingRobot` sind Roboter mit unterschiedlichen Bewegungsmöglichkeiten.
- `TinCan` kann sowohl laufen als auch fliegen und implementiert deshalb die beiden Interfaces `WalkingRobot` und `FlyingRobot`.



Die folgenden Definitionen setzen diese Konstruktion um.

```
interface Machine {
    void talk() default {
        throw new RuntimeException("I won\'t talk to you");
    }
```

```
}

interface Robot extends Machine {
    void talk() default {
        System.out.println("\"Blah\"");
    }
}

interface WalkingRobot extends Robot {
}

interface FlyingRobot extends Robot {
}

class TinCan implements WalkingRobot, FlyingRobot {
    public static void main(String... args) {
        Machine tinCan = new TinCan();
        tinCan.talk();
    }
}
```

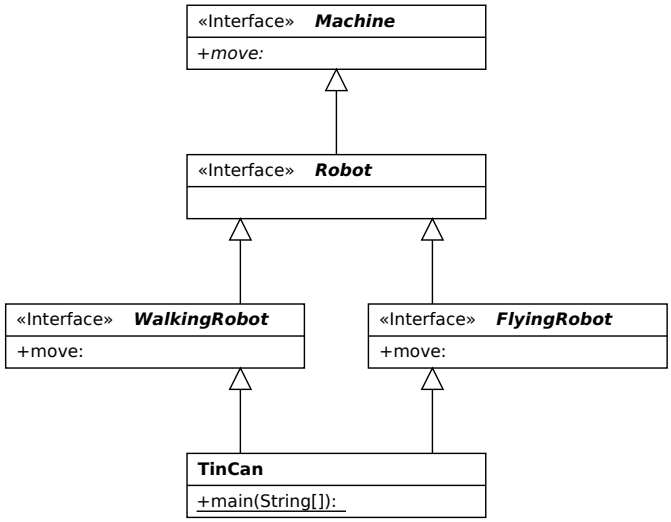
Listing 5.51: Eindeutige Default-Methode bei verschiedenen Interfaces.

Der Compiler übersetzt diesen Quelltext klaglos, weil er eine eindeutige Default-Methode findet. Das Programm ruft die `talk`-Methode von `Robot` auf:

```
$ java TinCan
"Blah"
```

Kollidierende Default-Methoden

Das Bild ändert sich mit Default-Methoden in den beiden Interfaces `WalkingRobot` und `FlyingRobot`:



```
interface Machine {
    void move();
}

interface Robot extends Machine {
}

interface FlyingRobot extends Robot {
    void move() default {
        System.out.println("Flap-flap ...");
    }
}

interface WalkingRobot extends Robot {
    void move() default {
        System.out.println("Hop, hop, ...");
    }
}

class TinCan implements WalkingRobot, FlyingRobot {
    public static void main(String... args) {
        Machine tinCan = new TinCan();
        tinCan.move();
    }
}
```

Listing 5.52: Sich widersprechende Default-Methoden in zwei verschiedenen Interfaces.

Jetzt stößt der Compiler auf kollidierende Implementierungen und bricht mit einem Fehler ab:

```
$ javac TinCan.java
TinCan.java: error: class TinCan inherits unrelated defaults for move()
    from types WalkingRobot and RollingRobot
```

Das Problem verschwindet, wenn

1. entweder eine der `walk`-Methoden in `WalkingRobot` und `FlyingRobot` gelöscht wird oder
2. `TinCan` selbst eine `walk`-Methode definiert.

Diamond-Problem der Mehrfachvererbung

Das hier dargestellte Problem ist auch als *Diamond-Problem* der Mehrfachvererbung bekannt. Dieses Problem kann bei größeren Typhierarchien schwer überschaubare Konsequenzen haben, insbesondere wenn ein Teil der beteiligten Typen nur in

übersetzter Form vorliegt und der Quelltext fehlt. Verschiedene Programmiersprachen gehen unterschiedlich damit um. Die Regelung von Java ist verhältnismäßig geradlinig.

Die Umsetzung in Java profitiert davon, dass Default-Methoden keine Daten mit sich bringen, sondern nur Funktionalität (siehe Seite 346). Zum Beispiel stellt sich daher die Frage nicht, *wie oft* ein Interface mit Default-Methoden auf unterschiedlichen Wegen „ererbt“ wird. Infolgedessen ist auch das Problem einer Reihenfolge oder Rangordnung gegenstandslos.

5.6.5 Evolution von Interfaces

Interfaces kaum veränderlich bis Java 7

Bis Java 7 sind einmal definierte Interfaces praktisch in Marmor gemeißelt. Änderungen an Interfaces haben in der Regel fatale Auswirkungen auf alle implementierenden Klassen, ebenso wie auf ihre Anwendungen. Entsprechend heikel ist der Entwurf von Interfaces.

Default-Methoden erlauben nachträgliche Erweiterung

Default-Methoden mildern dieses Problem deutlich ab, weil implementierende Klassen und ihre Anwendungen nachträgliche Erweiterungen eines Interface unbeschadet überstehen können.

Als Beispiel dient eine neue, vorher nicht im Interface `Rectangle` (Listing 5.44) enthaltene Methode `largerThan`. Sie liefert genau dann `true`, wenn ein Rechteck echt breiter und echt höher als ein zweites Rechteck `that` ist. `largerThan` wird als Default-Methode eingefügt:

```
boolean largerThan(Rectangle that) default {
    return getWidth() > that.getWidth()
           && getHeight() > that.getHeight();
}
```

Listing 5.53: Nachträglich in ein bereits verwendetes Interface eingefügte Default-Methode.

Die beiden bestehenden Implementierungen des Interface, `SimpleRectangle` (Listing 5.45) und `Square` (Listing 5.46), wurden bereits vor der Erweiterung definiert und wissen nichts von dieser nachträglichen Änderung! Dennoch müssen sie nicht angepasst werden und können ohne Neu-Übersetzung in einer neuen Anwendung, die mit der nachgeschobenen Methode arbeitet, weiter genutzt werden:

```
for(Rectangle rectangle: rectangles)
    for(Rectangle that: rectangles)
```

```
System.out.println(rectangle.largerThan(that));
```

Listing 5.54: Aufruf einer Methode, die nachträglich als Default-Methode in ein Interface eingefügt wurde.

Erweiterungen möglich, Änderungen und Kürzungen nicht

Default-Methoden erlauben einen viel flexibleren Umgang mit Interfaces als bisher. *Beliebige* Umbauten in Interfaces sind natürlich weiterhin nicht möglich:

Neue Methoden
: sind als Default-Methoden unkritisch für bestehende Implementierungen des Interface.
Bestehende Anwendungen sind von neuen Methoden nicht betroffen.

Gelöschte Methoden
: sind kein Problem für implementierende Klassen, weil diese dann einfach überzählige Methoden enthalten, die das Interface nicht (mehr) fordert.
Bestehende Anwendungen werden durch gelöschte Methoden unbrauchbar, weil sie nicht mehr existierende Methoden aufrufen.

Geänderte Methoden
: sind mit und ohne Default-Methoden für implementierende Klassen und Anwendungen tödlich.
Werden beispielsweise `getWidth` und `getHeight` in `Rectangle` (Listing 5.44) so geändert, dass sie ein Ergebnis des Aufzählungstyps

```
enum Size {Small, Medium, Large}
```

liefern, statt wie bisher `double`, dann ruiniert das unweigerlich alle Implementierungen und Anwendungen des Interface.

5.6.6 ABCs und Default-Methoden

Default-Methoden konkurrieren mit ABCs

Default-Methoden stellen Code zur Verfügung, den konkrete Klassen einfach übernehmen oder neu definieren können. Einen ganz ähnlichen Zweck erfüllen viele abstrakte Basisklassen, die einen Teil der Methoden eines (in der Regel) umfangreichen Interface auf der Grundlage einiger weniger abstrakter Methoden definieren.

Ein Beispiel ist die ABC `AbstractCollection`, die das Interface `Collection` implementiert. `Collection` definiert mehr als ein Dutzend Methoden, die eine konkrete Klasse alle realisieren müsste. Die ABC `AbstractCollection` vereinfacht das deutlich, weil sie für alle (bis auf zwei) `Collection`-Methoden Implementierungen zur Verfügung stellt. Eine konkrete Klasse, die von `AbstractCollection` erbt, muss

im einfachsten Fall nur *zwei* Methoden beisteuern. Natürlich steht es einer solchen Klasse frei, aus Performance-Gründen mehr als nur die zwei Pflichtmethoden zu redefinieren.

Alle konkreten Methoden der ABC `AbstractCollection` könnten als Default-Methoden in das Interface `Collection` verschoben werden. Die ABC `AbstractCollection` wäre dann überhaupt nicht mehr nötig.

Vor- und Nachteile von Default-Methoden

Default-Methoden leisten im Grunde das Gleiche wie konkrete Methoden in ABCs. Anders als ABCs bleiben Default-Methoden aber für Implementierungen unsichtbar. Insbesondere schließen sie nicht die Ableitung von einer anderen Basisklasse aus.

Im Gegenzug haben Default-Methoden keinen Zugriff auf Objektvariablen, weil sie in Interfaces definiert sind. Interfaces können zwar Variablen enthalten, diese sind aber in jedem Fall nur öffentliche Konstanten.

Zusammenfassung

- **Statisch geschachtelte Klassen** leben in einem gemeinsamen Gültigkeitsbereich und **teilen private Elemente** wie Variablen und Methoden.
- Objekte **innerer Klassen** sind unverbrüchlich an ein Objekt der äußeren Klasse gekoppelt.
- Objekte **lokaler Klassen** kapseln ihre lexikalische Definitionsumgebung in **Closures**.
- Closures sind **Funktionsobjekte,** das heißt Objekte mit Code als Wert.
- Java erlaubt **lokale Variablen** in einer Closure nur mit dem **Modifier** `final`.
- **Anonyme Klassen** sind namenlose lokale Klassen, die ein Interface implementieren oder eine Basisklasse ableiten. Ihre Objekte sind Closures, wie bei lokalen Klassen.
- **Lambda-Ausdrücke** sind eine kompakte Schreibweise von Funktionsobjekten. Sie müssen an Variablen eines Funktionsinterface zugewiesen werden.
- Ein **Funktionsinterface** hat genau eine Methode, die der Lambda-Ausdruck implementiert.
- Die **Laufzeitbibliothek** bietet viele **Anwendungsmöglichkeiten** für Lambda-Ausdrücke. Das erfordert allerdings Änderungen an lange etablierten Interfaces.
- **Default-Methoden** erlauben nachträgliche Erweiterungen von Interfaces, wobei bestehende Implementierungen intakt bleiben.
- Default-Methoden konkurrieren nicht mit Vererbung. Sie können aber unter gewissen Voraussetzungen die **Rolle von ABCs** übernehmen.

Aufgaben

Aufgabe 1: Statische Factory

Ein exklusiver Club hat nicht allzu viele Mitglieder. Eine Anwendung zur Clubverwaltung geht davon aus, dass jedes Objekt der Klasse `ClubMember` den Namen eines Mitglieds und ein paar andere Angaben kapselt. Man könnte `ClubMember` als Aufzählungstyp mit einem Element für jedes Clubmitglied definieren, aber dann müsste die Anwendung bei Ein- und Austritten neu übersetzt werden.

Die Name der Clubmitglieder stehen, einer pro Zeile, in einer Textdatei `members.txt`. Diese Datei wird beim Programmstart gelesen. Es gibt nur `ClubMember`-Objekte mit diesen Namen und sonst keine.

Definieren Sie die Klasse `ClubMember` mit einem privaten Konstruktor. Definieren Sie innerhalb von `ClubMember` eine statisch geschachtelte Klasse `Factory` mit der Methode

```
public Member  make(String name)
```

die ein neues Objekt liefert, falls `name` den Namen eines Clubmitglieds nennt. Die Methode liefert ein Objekt mit diesem Namen oder `null`, also kein Objekt, wenn der Name nicht in `members.txt` steht.

Das folgende Programm versucht, für jedes Kommandozeilenargument ein `ClubMember`-Objekt zu erzeugen. Wenn das misslingt, gibt es eine Warnung aus. Andernfalls fügt es das Objekt in eine Liste ein, die am Ende ausgegeben wird.

```
import java.io.*;
import java.util.*;

public class ClubMain {
    public static void main(String... args) {
        List<ClubMember> list = new ArrayList<>();
        ClubMember.Factory factory = new ClubMember.Factory();
        for(String arg: args) {
            ClubMember member = factory.make(arg);
            if(member == null)
                System.out.println("Not a member: " + arg);
            else
                list.add(member);
        }
        System.out.println(list);
    }
}
```

Listing 5.55: Testprogramm für eine Factory-Methode.

Die Datei `members.txt` hat beispielsweise den folgenden Inhalt:

```
$ cat members.txt
DagobertDuck
MacMoneysac
KlaasKlever
```

`ClubMain` kann Objekte mit diesen drei Namen erzeugen, aber keine anderen. Bei Änderungen an der Liste kann das Programm unverändert bleiben.

```
$ java ClubMain DagobertDuck DonaldDuck KlaasKlever Panzerknacker
Not a member: DonaldDuck
Not a member: Panzerknacker
[DagobertDuck, KlaasKlever]
```

Es gibt auch andere Lösungswege:

- Die Klassen `Factory` und `ClubMember` können getrennt definiert und beide in ein eigenes Package verschoben werden. Der Zugriffsschutz des `ClubMember`-Konstruktors muss auf Package-Ebene gelockert werden.
- Die Klasse `Factory` kann aufgelöst und ihre Elemente können der äußeren Klasse `ClubMember` zugeschlagen werden. Allerdings würde das die Klasse `ClubMember` mit Code aufblähen, die dort nichts zu suchen hat.
- Die Logik der `Factory`-Methode `make` kann in den `ClubMember`-Konstruktor verschoben werden. Der Preis wäre der gleiche wie bei der vorhergehenden Alternative. Darüber hinaus kann ein Konstruktor nicht `null` liefern, um ein Problem zu signalisieren.

Ableiten einer statisch geschachtelten Klasse

Die Klasse `Factory` lädt die Textdatei `members.txt` beim Programmstart und beachtet sie dann nicht mehr. Leiten Sie innerhalb von `ClubMember` von `Factory` eine weitere Klasse `LiveFactory` ab. Die neue Klasse verwendet immer den gerade aktuellen Inhalt der Datei `members.txt`. Das bedeutet, dass Änderungen an `members.txt` *während* der Laufzeit einer Anwendung sofort wirksam werden. Für diesen Zweck ist die Methode `Files.getLastModifiedTime` (Seite 96) von Nutzen. Kopieren Sie keinen Code der Basisklasse, aber nutzen Sie die Möglichkeit, auf private Elemente zuzugreifen.

Verwenden Sie das folgende, gegenüber `ClubMain` (Listing 5.55) verlangsamte Testprogramm, das alle drei Sekunden ein Objekt erzeugt:

```
import java.util.*;

public class SlowClubMain {
    public static void main(String... args) throws InterruptedException {
        List<ClubMember> list = new ArrayList<>();
        ClubMember.Factory factory = new ClubMember.LiveFactory();
        for(String arg: args) {
            // wie in der main-Methode der Basisklasse ...
            Thread.sleep(3000);
        }
        System.out.println(list);
    }
}
```

Listing 5.56: Testprogramm für eine Factory-Klasse, die auf Änderungen im Filesystem reagiert.

Starten Sie das Programm wie folgt:

```
$ java SlowClubMain DagobertDuck DonaldDuck KlaasKlever Panzerknacker
Not a member: DonaldDuck
...
```

Sobald das Programm diese Zeile ausgegeben hat, verlängert ein Hacker (das sind Sie selbst) in einem anderen Terminal die Datei `members.txt` mit dem folgenden Kommando um den Namen `Panzerknacker`:

```
$ echo Panzerknacker >> members.txt
```

`SlowClubMain` sollte nur noch ausgeben:

```
...
[DagobertDuck, KlaasKlever, Panzerknacker]
```

Wiederholen Sie den Testlauf, löschen Sie aber diesmal nach der ersten Ausgabe die Datei `members.txt`. Jetzt sollte die Anwendung ausgeben:

```
...
Not a member: DonaldDuck
Not a member: KlaasKlever
Not a member: Panzerknacker
[DagobertDuck]
```

Aufgabe 2: Teilmatrix

Objekte innerer Klassen enthalten eine Referenz auf das Objekt der äußeren Klasse, in dessen Kontext sie entstanden sind. Diese Referenz ist verborgen und unveränderlich. Sie eignet sich zur Modellierung von „Sichten“, die einen neuen Zugang zu bestehenden Daten bieten.[36]

Das folgende Interface definiert eine einfache Schnittstelle für Matrizen ganzer Zahlen mit der einzigen Operation `add`:

```
public interface Matrix {
    int getWidth();

    int getHeight();

    int get(int row, int column);

    Matrix add(Matrix that);
}
```

Listing 5.57: Einfache Schnittstelle für Matrix-Klassen.

Zeilen- und Spaltenpositionen sind 0-basiert. Matrizen sind veränderlich: Die Methode `add` addiert die andere, gleich große Matrix `that` zu `this` und liefert das modifizierte `this` zurück.

Definieren Sie eine Klasse `Array2DMatrix`, die `Matrix` geradlinig auf der Basis eines zweidimensionalen `int`-Arrays implementiert. Geben Sie der Klasse eine `toString`-Methode, die sich auf `Arrays.deepToString` stützt. `Array2DMatrix` hat zwei Konstruktoren:

`Array2DMatrix(int h, int w)`
: Erzeugt eine Matrix mit `h` Zeilen Höhe und `w` Spalten Breite, deren Elemente alle 0 sind.

`Array2DMatrix(int h, int... values)`
: Erzeugt eine Matrix mit `h` Zeilen Höhe und füllt sie zeilenweise mit den Elementen `values`. Die Breite ergibt sich aus `h` und der Anzahl Argumente `values`.

Definieren Sie weiter in `Array2DMatrix` eine innere Klasse `Sub2DMatrix`, die einen *Ausschnitt* der Matrix der äußeren Klassen repräsentiert. `Sub2DMatrix` bietet den folgenden Konstruktor an:

[36] In der Bibliothek ist beispielsweise `Entry` eine Sicht auf eine `Map`.

`Sub2DMatrix(int top, int left, int h, int w)`
: Ein Ausschnitt der äußeren Matrix, dessen oberer Rand in Zeile `top` und dessen linker Rand in Spalte `left` liegt. Der Ausschnitt ist `h` Zeilen hoch und `w` Spalten breit.

`Sub2DMatrix` ist selbst eine Matrix, implementiert also ebenfalls das Interface `Matrix`. Eine `Sub2DMatrix` speichert aber *keine eigenen* Elemente, sondern merkt sich nur Lage und Größe des Ausschnitts und arbeitet ansonsten mit den Daten des äußeren `Array2DMatrix`-Objekts.

Änderungen an einer `Sub2DMatrix` wirken sich auf die äußere Matrix aus und umgekehrt. Eine `Sub2DMatrix` ist eine **Sicht** auf eine `Array2DMatrix`.

Aufgabe 3: Anonyme Collection

Das generische Interface `Collection` steckt die gemeinsame Funktionalität aller Sammlung von Einzelwerten im Collection-Framework ab. `Collection` umfasst allerdings 15 Methoden und erfordert damit einigen Aufwand zur Implementierung. Zur Vereinfachung enthält die ABC `AbstractCollection` Implementierungen der meisten der Methoden auf der Grundlage von nur zwei abstrakten Methoden:[37]

`abstract Iterator<E> iterator()`
: Iterator über die Elemente, dessen Methode `remove` eine `UnsupportedOperationException` werfen darf.

`abstract int size()`
: Anzahl Elemente in der Collection.

Eine konkrete Klasse, die noch diese beiden Methoden beisteuert, definiert eine komplette, wenn auch unveränderliche Collection.[38]

Das folgende Programm verarbeitet die Kommandozeilenargumente als unveränderliche Collection von Stringpaaren, wobei immer zwei aufeinanderfolgende Kommandozeilenargumente ein Paar vom Typ `Pair` bilden. Die entsprechende Collection-Klasse wird nur einmal gebraucht und kann daher als anonyme Klasse definiert werden. Das Gleiche gilt für die konkrete Implementierung des Interface `Pair`. Ergänzen Sie den fehlenden Code um anonyme Klassendefinitionen:

[37] Die Methoden in `AbstractCollection` sind sicher keine Performance-Wunder, aber logisch korrekt.

[38] Für eine veränderliche Collection müssen zwei weitere Methoden definiert werden, nämlich `add` in der Collection-Klasse selbst und `remove` im Iterator.

```
import java.util.*;

public class StringArrayCollectionMain {
    interface Pair extends Comparable<Pair> {   // automatisch public und static
        String getFirst();

        String getSecond();
    }

    public static void main(final String... args) {
        Collection<Pair> collection = new AbstractCollection<Pair>() {
            // Ergänzen Sie hier den fehlenden Code ...
        };
        Pair first = collection.iterator().next();
        System.out.println(Collections.frequency(collection, first));
        System.out.println(Collections.max(collection));
        System.out.println(Collections.min(collection));
    }
}
```

Listing 5.58: Programm, das die Kommandozeilenargumente als Collection von Strings verarbeitet.

Das Programm erwartet eine gerade Anzahl von Kommandozeilenargumenten und sollte folgendermaßen funktionieren:

```
$ java StringArrayCollectionMain sim sala bim abra ka dabra sim sala
2
sim/sala
bim/abra
```

Die Lösung erfordert eine vergleichsweise ausladende anonyme Klassendefinition. Zu Übungszwecken ist das zwar hinnehmbar, im Allgemeinen sind anonyme Klassen aber für kurze und überschaubare Definitionen gedacht.

Kapitel

6 Nebenläufigkeit

Lernziele

In diesem Kapitel lernen Sie

- dass **Nebenläufigkeit** in Java in Form der Klasse `Thread`[1] fest in der Sprache verankert ist und wie Sie damit in einem Programm parallel ablaufende Threads starten können.
- wie sich Threads über **Interrupts** auf einfache Art miteinander verständigen können.
- wie die **JVM Threads verwaltet** und warum Threads so wichtig für moderne Multiprozessor-Systeme sind.
- welche Probleme auftreten, wenn Threads gleichzeitig *dieselben* Daten manipulieren und wie Sie mit **Synchronisation** für Ordnung sorgen können.
- wie Sie mit **bedingtem Warten** mehrere Threads so koordinieren können, dass sie gemeinsam eine Aufgabe lösen und dabei effizient mit der verfügbaren Rechenleistung umgehen.

Für dieses Kapitel kann man sich ein Programm wie eine Küche in einem Restaurant vorstellen. Die Aufgabe des Programms entspricht der Zubereitung eines mehr oder weniger aufwendigen Menüs. Variablen, Objekte und weitere Datenstrukturen wären die verschiedenen Küchenutensilien und -geräte, wie Zutaten, Geschirr und Öfen. Ein **Thread** ist in diesem Bild ein Koch, der in der Küche arbeitet.

In den bisher entwickelten Programmen gab es immer nur *einen* Thread, auch wenn darauf nicht ausdrücklich hingewiesen wurde. In diesem Kapitel geht es um Programme, die mit mehreren Threads arbeiten. Man nennt solche Programme auch „nebenläufig“, weil mehrere Vorgänge nebeneinander ablaufen.

[1] Der Begriff „Thread“ (Faden) spielt auf die Abfolge der Anweisungen an, die beim Programmablauf nacheinander ausgeführt werden. Diese Abfolge zieht sich durch den Quelltext wie ein Faden durch ein Labyrinth.

Bezogen auf das Küchenbeispiel bedeutet das, dass für die Zubereitung des gesamten Menüs nur ein einziger Koch zuständig ist. Das Ziel dieses Kapitels ist es im übertragenen Sinn, weitere Köche in der Küche zu beschäftigen, die zusammen an der Aufgabe arbeiten.

Daraus ergeben sich wesentliche Ziele der Nebenläufigkeit:

- Für ein aufwendiges Menü müsste sich ein einzelner Koch Vorspeise, Hauptgericht und Nachtisch nacheinander vornehmen. Mehrere Köche können sich dagegen gleichzeitig um die verschiedenen Gänge kümmern.
  Übertragen auf Software bedeutet das, dass ein nebenläufiges Programm verschiedene Teilaufgaben gleichzeitig bearbeiten und damit eine Aufgabe insgesamt schneller bewältigen kann als ein Programm mit einem einzigen Thread.[2]
- Auch bei einem Menü mit nur einem Gang sind mehrere Köche nützlich. Wenn ein Koch beispielsweise einen Topf voll Milch zum Kochen bringen will, muss er ihn im Auge behalten[3] und kann in dieser Zeit sonst nichts tun. Ein zweiter Koch kann währenddessen andere Vorbereitungen treffen.
  Bezogen auf Java heißt das, dass ein Programm, das nur einen einzigen Thread verwendet, bei einer blockierenden Operation (typischerweise I/O) tatenlos warten muss, bis sie zu einem Ende kommt. Ein nebenläufiges Programm kann dagegen in der Zwischenzeit andere Aufgaben erledigen.

Aber auch Probleme deuten sich hier schon an:

- Während jeder Koch sicher ein eigenes Messer besitzt, müssen sich die Köche wahrscheinlich größere Küchengeräte teilen. Dazu sind Absprachen nötig.
  Bezogen auf Programme heißt das, dass sich Threads über die Nutzung gemeinsamer Datenstrukturen einigen müssen (Kapitel 6.4).
- Je enger Köche zusammenarbeiten, desto kleinere Aufgaben sind möglich. Diese Aufgaben werden zunehmend voneinander abhängen. Beispielsweise muss erst eine bestimmte Soße fertiggestellt sein, bevor sie in anderen Speisen verwendet werden kann.
  In einem Programm müssen Threads, die aufeinander angewiesen sind, zeitlich koordiniert werden. Abhängige Threads müssen vorübergehend warten, bis andere Threads die vereinbarten Voraussetzungen geschaffen haben und eine Fortsetzung sinnvoll ist (Seite 6.6).
- Mit der Anzahl der beteiligten Köche steigt der Zeitaufwand, der zur Abstimmung nötig ist. Es kommt der Punkt, an dem die Köche nichts anderes mehr tun, als sich abzusprechen.

2 Das setzt auch voraus, dass der Rechner über mehrere CPUs beziehungsweise CPU-Cores verfügt.

3 Erfahrungsgemäß kocht Milch vorzugsweise dann über, wenn man kurz den Blick abwendet.

Threads müssen verwaltet werden. Wenn der Verwaltungsaufwand überhand nimmt, lohnen sich weitere Threads nicht mehr. Es gibt eine Rentabilitätsgrenze.

6.1 Paralleler Programmablauf

6.1.1 Basisklasse `Thread`

Ein Java-Programm startet immer mit einem ersten Thread, der `main` ausführt. Weitere Threads entstehen nicht automatisch, sondern müssen explizit erzeugt und gestartet werden.

`Thread`-Methoden

Der Schlüssel zu neuen Threads ist die Klasse `Thread` mit den beiden folgenden Methoden, die weder einen Parameter haben, noch ein Ergebnis liefern:

Wichtigste Methoden der Basisklasse `Thread`

`void start()`
: Ruft einen neuen Thread ins Leben.

`void run()`
: Enthält den Code, den der neue Thread ausführt. In der Klasse `Thread` ist diese Methode noch leer und zur Redefinition bestimmt.

Begriff Thread

Der Begriff „Thread“ taucht in diesem Kapitel in zwei verwandten, aber trotzdem unterschiedlichen Bedeutungen auf:

„Thread“ als Typ und als Programmablauf

- Die **Klasse** `Thread` definiert einen Typ mit Konstruktoren, Methoden und so weiter. Von diesem Typ lassen sich beispielsweise Objekte erzeugen und neue Klassen ableiten. In dieser Bedeutung bezeichnet `Thread` einen Typ, der in erster Linie für den Compiler interessant ist.
- Zur Laufzeit bezeichnet „Thread“ einen **Ablauf** im Programm, also ablaufenden Bytecode. Dieser Begriff von Thread ist für die virtuelle Maschine interessant.

Zur Unterscheidung ist das Wort „`Thread`“ in `Codeschrift` gesetzt, wenn Typen und Klassen gemeint sind. Das Wort „Thread“ in normaler Textschrift steht für

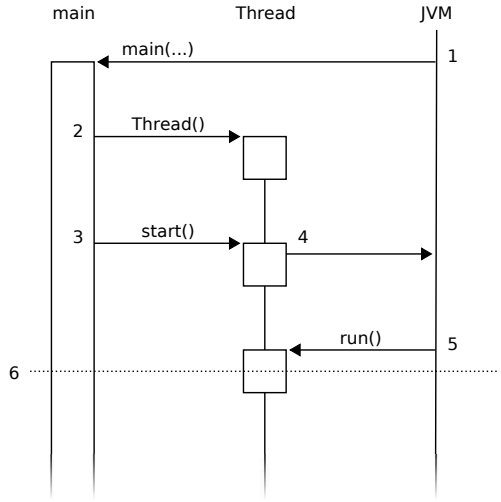
einen Kontrollfluss, also ausgeführten Code zur Laufzeit. Die Klasse `EmptyThread` ist das vielleicht einfachste Programm, das einen neuen Thread startet:[4]

```
public class EmptyThread {
    public static void main(String... args) {
        Thread thread = new Thread();
        thread.start();
    }
}
```

Listing 6.1: Start eines Threads, der nichts macht.

Mittelbarer Aufruf von `run` durch die JVM

So einfach dieser Quelltext aussieht, zeigt das Programm doch einen komplexen Ablauf, den das folgende Sequenzdiagramm veranschaulicht:



1. Die JVM startet das Programm mit dem Aufruf von `main`.
2. `main` erzeugt mit dem Default-Konstruktor ein neues `Thread`-Objekt.
3. `main` ruft die Methode `start` des neuen `Thread`-Objekts auf.
4. `start` weist die JVM an, umgehend einen neuen Thread zu initialisieren. `start` kehrt sofort zurück. Insbesondere wartet `start` nicht, bis der neue Thread läuft.
5. Nach kurzer Zeit erzeugt die JVM einen *neuen Thread*, der die Methode `run` ausführt.
6. `main` und `run` laufen *gleichzeitig*, wenn auch in diesem Beispiel nur für kurze Zeit.

[4] Das Programm könnte noch weiter auf die Anweisung `new Thread().start();` verkürzt werden, wenn die Variable `Thread th` eingespart wird.

6.1.2 Start eines neuen Threads

Die Klasse `Thread` ist zum Ableiten vorgesehen.

Ableiten von `Thread` mit neuem Verhalten

- Sie definiert zwar selbst eine Methode `run`, allerdings mit leerem Rumpf. Eine Anwendung wird also von `Thread` eine neue Klasse *ableiten* und dabei `run` mit einem sinnvollen Verhalten *redefinieren*.[5]
- Die Methode `start` löst die schwarze Magie aus, die zum Start eines neuen Threads führt. `start` kann mit den „Bordmitteln“ von Java nicht implementiert werden. Eine Redefinition ist nicht sinnvoll.

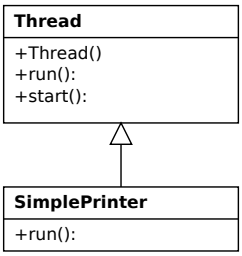
`run` ist eine Callback-Methode. Eine Anwendung ruft sie *nicht selbst* auf, sondern lässt `run` mittelbar von der JVM aufrufen. Den Auftrag dazu erteilt `start`.[6]

Die folgende Klasse leitet `Thread` ab und redefiniert die `run`-Methode mit einer Ausgabe.

```
public class SimplePrinter extends Thread {
    public void run() {
        System.out.println("#");
    }

}
```

Listing 6.2: Thread, der ein Zeichen ausgibt.

Die folgende Skizze zeigt die Konstruktion:



Die `main`-Methode erzeugt ein `SimplePrinter`-Objekt, ruft die `start`-Methode auf und gibt nach der Rückkehr von `start` selbst noch ein Zeichen aus:

[5] Alternativ kann auch das Interface `Runnable` implementiert werden (siehe Seite 375). Das läuft aber letztlich auf praktisch den gleichen Code hinaus.

[6] Eine Anwendung kann selbst direkt `run` aufrufen. Der Aufruf unterscheidet sich dann aber durch nichts von einem gewöhnlichen Methodenaufruf. Der Aufrufer wartet wie bei jedem Methodenaufruf auf die Rückkehr von `run` und fährt dann fort. Insbesondere wird kein neuer Thread erzeugt.

```
public static void main(String... args) {
    Thread thread = new SimplePrinter();
    thread.start();
    System.out.println(":");
}
```

Listing 6.3: Start eines Threads und anschließende Ausgabe eines Zeichens.

Startet man das Programm, so erhält man die beiden Ausgaben:

```
:
#
```

Das ist nicht sehr beeindruckend. Von den beiden Threads, `SimplePrinter` und `main` selbst, ist nichts zu bemerken. Das ändert sich, wenn man die Ausgaben in Schleifen einige Male wiederholt.[7]

```
import static java.lang.System.*;

public class LoopPrinter extends Thread {
    public void run() {
        for(int i = 0; i < 1000; i++)
            out.print("#");
    }

    public static void main(String... args) {
        new LoopPrinter().start();
        for(int i = 0; i < 1000; i++)
            out.print(":");
    }
}
```

Listing 6.4: Zwei Schleifen in verschiedenen Threads, die beide Zeichen ausgeben.

Nachweis der parallelen Ausführung

Dieses Programm gibt eine Mischung[8] beider Zeichen aus, die die gleichzeitig laufenden Threads produzieren.[9]

```
$ java LoopPrinter
```

[7] In diesem Programm ist die `main`-Methode direkt in der `Thread`-Klasse selbst definiert. Das ist eher ungewöhnlich und dient hier nur zur Vereinfachung des Beispielcodes. In der Regel stecken `run` und `main` nicht in derselben Klasse.

[8] Die hier abgedruckte Ausgabe ist nur ein Beispiel. Wenn Sie das Programm selbst ausprobieren, wird die Reihenfolge der Zeichen anders aussehen.

[9] Mit `print` statt `println` erhält man eine kompaktere Ausgabe, die allerdings in einer einzigen Zeile steht. Zum Abdruck wurden hier nachträglich Zeilenumbrüche eingefügt.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::###########
##################################################################################################
####################################:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::#########################
##################################################################################################
##################:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::##################
(und so weiter)
```

Zusätzliche Kontrollausgaben am Anfang und am Ende der Schleifen machen den parallelen Ablauf klarer sichtbar:

Zeitliche Verschiebung von Start und Ende

```
import static java.lang.System.*;

public class LoopPrinter extends Thread {
    public void run() {
        out.println("run loop start");
        for(int i = 0; i < 1000; i++)
            out.print("#");
        out.println("run loop end");
    }

    public static void main(String... args) {
        new LoopPrinter().start();
        out.println("main loop start");
        for(int i = 0; i < 1000; i++)
            out.print(":");
        out.println("main loop end");
    }
}
```

Listing 6.5: Parallele Ausgabeschleifen mit Protokollausgaben am Anfang und am Ende.

```
$ java LoopPrinter
main loop start
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::run loop start
##################################################################################################
####################################:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::##############
(und so weiter)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
##################################################################################################
####################################run loop end
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::main loop end
```

Selbststartende Threads

In der Regel startet eine Anwendung ein neu erzeugtes `Thread`-Objekt auch umgehend mit `start`. Zur Vereinfachung kann eine `Thread`-Klasse die `start`-Methode

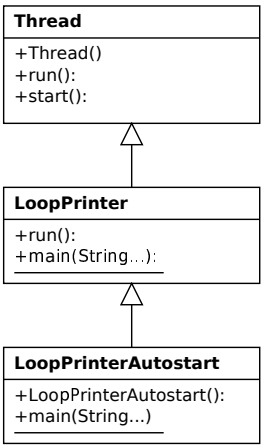
Gefährliche Bequemlichkeit

auch selbst im eigenen Konstruktor aufrufen und sich damit selbst starten, wie zum Beispiel:

```
public class LoopPrinterAutostart extends LoopPrinter {
    public LoopPrinterAutostart() {
        start();
    }

    public static void main(String... args) {
        new LoopPrinterAutostart(); // kein start-Aufruf
        for(int i = 0; i < 1000; i++)
            System.out.print(":");
    }
}
```

Listing 6.6: Thread, der sich im Konstruktor selbst startet.

Der `LoopPrinterAutostart`-Konstruktor ruft erst implizit den vom Compiler automatisch definierten Basisklassen-Default-Konstruktor auf:



Die Anwendung muss jetzt nur ein `Thread`-Objekt erzeugen, das ungefragt sofort loslegt.

Dieser „Luxus“ ist aber im Zusammenhang mit Vererbung riskant: Der Konstruktor einer weiter abgeleiteten Klasse ist möglicherweise noch nicht abgeschlossen, wenn der `start`-Aufruf schon mittelbar den `run`-Aufruf auslöst. `run` könnte dann ein erst teilweise initialisiertes Objekt vorfinden, mit kaum absehbaren Folgen. Nur bei nicht ableitbaren, mit `final` markierten Klassen fällt dieses Risiko weg.

Das folgende Programm demonstriert das Problem:

```
import static java.lang.System.*;

public class LoopPrinterBrokenAutostart extends LoopPrinterAutostart {
    private final int loops;

    public LoopPrinterBrokenAutostart() {
        loops = Integer.parseInt(console().readLine("Number of loops? "));
    }

    public void run() {
        out.println("run loop start");
        for(int i = 0; i < loops; i++)
            out.print("#");
        out.println("run loop end");
    }

    public static void main(String... args) {
        new LoopPrinterBrokenAutostart();
        out.println("main loop start");
        for(int i = 0; i < 1000; i++)
            out.print(":");
        out.println("main loop end");
    }
}
```

Listing 6.7: Basisklassenkonstruktor mit zu frühem Thread-Start.

Beim Programmaufruf startet der Basisklassenkonstruktor den Thread sofort. Die JVM ruft daraufhin `run` bereits auf, *bevor* die Objektvariable `loops` mit einer Benutzereingabe initialisiert werden kann. Zu diesem Zeitpunkt hat `loops` noch den Defaultwert 0, sodass die Thread-Schleife überhaupt nicht durchlaufen wird. Bei der Rückkehr des `LoopPrinterBrokenAutostart`-Konstruktors ist der Thread schon beendet.

```
$ java LoopPrinterBrokenAutostart
run loop start
run loop end
Number of loops? 100
main loop start
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
(und so weiter)
::::::::::::::::::::::main loop end
```

6.1.3 Eigenschaften von Threads

Gepufferte Ausgabe

Die Ausgabe von `LoopPrinter` zeigt abwechselnde Zeichenblöcke unterschiedlicher Länge, keine abwechselnden Einzelzeichen. Der Grund dafür ist die Pufferung

der Ausgabe, die eine rentable Anzahl einzeln ausgegebener Zeichen ansammelt, bevor sie im Paket zum Bildschirm geschickt werden.

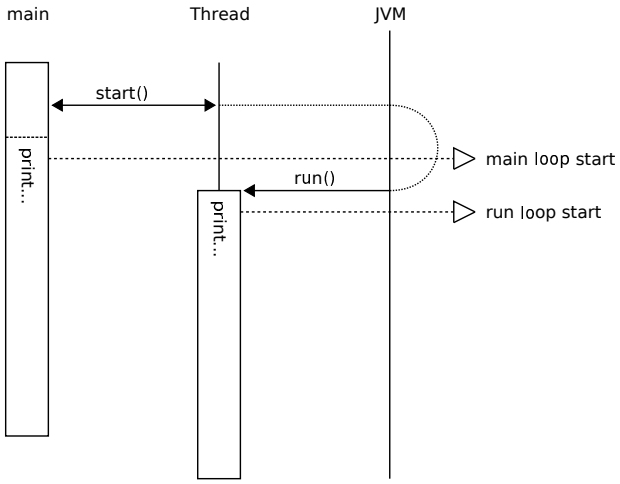
Verzögerter Thread-Start

Genauer Zeitpunkt eines Thread-Starts unbestimmt

In aller Regel beobachtet man beim Start von `LoopPrinter` als erste Ausgabe:

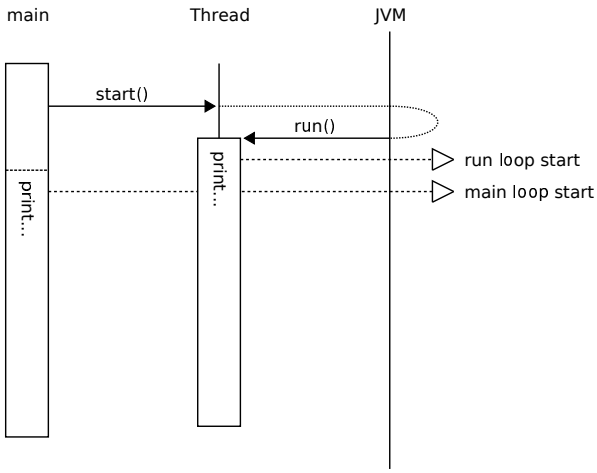
```
main loop start
```

`main` ruft zwar erst `start` auf und gibt anschließend diesen Text aus, aber die JVM braucht etwas Zeit, um den neuen Thread zu arrangieren und zu starten. In dieser Zeit läuft `main` schon weiter und beginnt wahrscheinlich auch schon die Schleife abzuarbeiten.



Es ist allerdings unbestimmt, wie lange nach dem `start`-Aufruf die JVM den Thread tatsächlich aktiviert. Im Extremfall könnte die erste Thread-Ausgabe sogar schon *vor* der ersten `main`-Ausgabe erscheinen.[10]

[10] Ebenso könnte der Thread erst starten, nachdem `main` komplett fertig ist.



Die Schleifenzahl von 1000 in `LoopPrinter` reicht meistens aus, um beide Threads parallel zum Arbeiten zu bringen. Aber das ist nur ein Schätzwert. Bei einem kleinere Schleifenzähler ist es möglich, dass `main` schon endet, bevor der abgespaltene Thread überhaupt anfängt zu arbeiten.

Nicht deterministischer Ablauf

Zeitlicher Versatz zwischen Threads nicht vorhersagbar

Jeder Start von `LoopPrinter` produziert eine andere Abfolge von Zeichen und Blöcken. Die beiden Threads laufen in diesem Beispiel ohne Kopplung ab. Sie verschieben sich zeitlich in kaum vorhersehbarer Weise gegeneinander. Die genaue Abfolge der ausgeführten Anweisungen muss als *nicht deterministisch* betrachtet werden.

Auch die Ablaufgeschwindigkeit der beiden parallel laufenden Ausgabeschleifen ist nicht vorhersehbar. Sie können im Takt bleiben oder sich gegenseitig überholen. Das bedeutet auch, dass die Schleifen nicht in der gleichen Reihenfolge enden müssen, in der sie starten.

`run` als normale Methode

Synchroner Aufruf von `run`

`run` ist, isoliert betrachtet, keine besondere Methode. Das Besondere an `run` ist lediglich die Art und Weise, wie die Methode *aufgerufen* wird.[11]

`run` kann auch als normale Methode benutzt werden. Der Aufruf läuft dann aber *synchron* ab: Der Aufrufer wartet dann, bis `run` zurückkehrt, und fährt erst dann fort. Ersetzt man in `LoopPrinter` die Anweisung

[11] Das Gleiche gilt beispielsweise auch für `main`.

```
new LoopPrinter(). start();
```

durch

```
new LoopPrinter(). run();
```

dann folgen die Ausgaben der beiden Schleifen getrennt nacheinander:

```
$ java LoopPrinter
run loop start
##########################################################################################
(und so weiter)
##########################################################################################run loop end
main loop start
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
(und so weiter)
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::main loop en
```

Mehrfacher `Thread`-Start

Threads laufen nur einmal ab

Ein Thread kann nur einmal gestartet werden. Ein zweiter Aufruf von `start` desselben `Thread`-Objekts führt zu einer Exception. Dabei spielt es keine Rolle, ob der Thread schon beendet ist oder noch läuft.[12]

Eine Thread-Klasse kann mehrfach instanziiert werden. Jedes einzelne Thread-Objekt für sich akzeptiert einen `start`-Aufruf, der einen Aufruf von `run` dieses Objekts nach sich zieht.

6.1.4 Ende eines Threads

Thread-Ende mit Rückkehr aus `run`

Ein Thread endet mit dem Verlassen der `run`-Methode. Das `Thread`-Objekt selbst bleibt auch danach noch bestehen und kann wie jedes andere Objekt weiter verwendet werden.

Es gibt keine Möglichkeit einen Thread ohne dessen Zutun zu beenden, also gewissermaßen „gewaltsam“ von außen zu stoppen.[13] In jedem Fall muss ein Thread

[12] Man kann sich einen Thread wie einen Sternwerfer vorstellen. Der Konstruktor erzeugt ihn und der `start`-Aufruf entzündet ihn. Nachdem der Sternwerfer abgebrannt ist, existiert er zwar noch, enthält aber nur noch Asche. Die kann man untersuchen, aber nicht mehr entzünden.

[13] Eine Ausnahme machen sogenannte Daemon-Threads (Seite 377), die am Programmende automatisch abgebrochen werden.

aus freien Stücken und Kraft eigener Logik die Arbeit einstellen.[14]

Kooperative Threads

Threads haben viele Möglichkeiten in Kontakt zu treten. In der Regel bittet ein Thread einen anderen darum, doch umgehend zu einem Ende zu kommen. Dafür bieten zum Beispiel Interrupts (siehe nächster Abschnitt, 6.2) eine einfache Möglichkeit.

Methode `join`

Warten auf das Ende eines anderen Threads

Noch einfacher als als mit Interrupts können Threads mit den Methoden

```
final void  join()
```

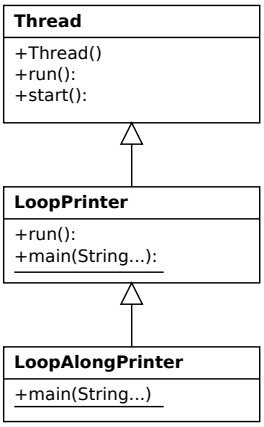
```
final void  join(long millis)
```

abgestimmt werden. Ein Aufruf von `join` an einen Thread kehrt erst dann zurück, wenn der Thread zum Ende gekommen ist. Die erste der beiden `join`-Methoden wartet auf unbestimmte Zeit, während die zweite spätestens nach der gegebenen Anzahl Millisekunden aufgibt und zurückkehrt. Wenn der betreffende Thread noch nicht gestartet[15] oder bereits beendet ist, kehrt `join` sofort zurück.

Die folgende Klasse `LoopAlongPrinter` ist ein `LoopPrinter` mit einer anderen `main`-Methode.

[14] Aus den Anfängen von Java haben bis heute die `Thread`-Methoden `stop` und `suspend` überwintert, mit denen ein Thread einen anderen beenden oder anhalten kann. Allerdings können dabei Objekte in einem inkonsistenten Zustand zurückbleiben. Die Methoden sind daher seit Langem *deprecated* (abgekündigt) und können in einer kommenden Java-Version kommentar- und ersatzlos fehlen. Es gibt sogar den kuriosen Fall der Methode `destroy`, die zwar definiert, aber nie implementiert wurde. Ihr Aufruf wirft immer einen `NoSuchMethodError`.

[15] Entscheidend ist hier der Aufruf von `start`, nicht der mittelbare Aufruf von `run`.



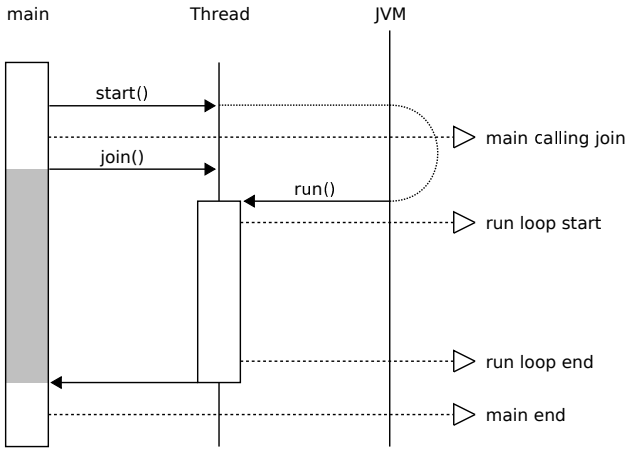
`main` startet einen Thread und wartet mit `join`, bis er endet.

```
public class LoopAlongPrinter extends LoopPrinter {
    public static void main(String... args) {
        try {
            Thread thread = new LoopAlongPrinter();
            thread.start();
            System.out.println("main calling join");
            thread.join();
            System.out.println("main end");
        }
        catch(InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
```

Listing 6.8: Thread, der mit `join` auf das Ende anderer Threads wartet.

Die Ausgabe `main end` erscheint *immer* zuletzt:

```
$ java LoopAlongPrinter
main calling join
run loop start
##########################################################################################
(und so weiter)
##########################################################################run loop end
main end
```



Methode `isAlive`

Test, ob ein anderer Thread läuft

Die Methode

```
final boolean  isAlive()
```

gibt Auskunft, ob ein anderer Thread gerade läuft. Sie liefert `true`, wenn der andere Thread gestartet, aber noch nicht aus `run` zurückgekehrt ist. Vor dem Start und nach Rückkehr aus `run` liefert die Methode `false`.

6.1.5 Interface `Runnable`

Alternative zur Ableitung von `Thread`

Die „einfache Vererbung“ von Java räumt einige heikle Probleme aus, hat aber auch Einschränkungen zur Folge. Besonders im Zusammenhang mit Threads steht man manchmal vor dem Dilemma, dass eine Klasse von einer Basisklasse erben und außerdem noch nebenläufig arbeiten, also auch noch von `Thread` abgeleitet werden müsste. Beides zusammen verbietet die einfache Vererbung.

Das Interface `Runnable` löst dieses Problem. Es verlangt nur die Methode

```
void  run()
```

mit – nicht ganz zufällig – genau der gleichen Signatur wie die entsprechende `Thread`-Methode.

Die Klasse `Thread` definiert, abgesehen vom bisher verwendeten Default-Konstruktor, eine Reihe weiterer Konstruktoren. Von diesen ist hier der Konstruktor

```
Thread(Runnable r)
```

mit einem `Runnable`-Parameter interessant.

Der `start`-Aufruf eines `Thread`-Objekts, das mit diesem zweiten Konstruktor initialisiert wurde, startet einen neuen Thread mit einem Aufruf der `run`-Methode von `r` anstelle der `Thread`-eigenen `run`-Methode. Entscheidend ist also die Wahl des `Thread`-Konstruktors.

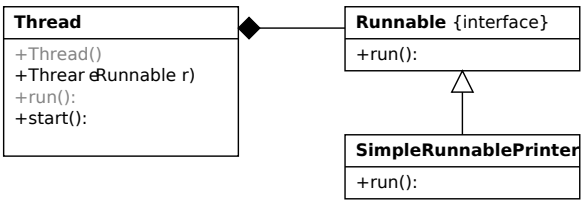
Das folgende Programm entspricht dem Programm `SimplePrinter` (Listing 6.2) fast genau, erbt aber nicht von `Thread`, sondern implementiert `Runnable`. Für die `run`-Methode hat das jedoch keine Auswirkung:

```
public class SimpleRunnablePrinter implements Runnable {
    public void run() {
        System.out.println("#");
    }

    public static void main(String... args) {
        Thread thread = new Thread(new SimpleRunnablePrinter());
        thread.start();
        System.out.println(":");
    }
}
```

Listing 6.9: Implementierung des Interface `Runnable` und Start eines Threads.

Thread mit abhängigem `Runnable`-Objekt

Jetzt sind zwei verschiedene Objekte im Spiel, wie das folgende Klassendiagramm zeigt:



In aller Kürze unterscheiden sich die beiden Varianten folgendermaßen:

- Ableiten von `Thread`

  ```
  class T  extends Thread {...}
  ...
  new  T().start();
  ```

- Implementieren von `Runnable`:

```
class T  implements Runnable {...}
...
new  Thread(new T()).start();
```

Die Entscheidung für Ableitung oder Implementierung ist in erster Linie eine Frage der Modellierung, hat aber keine nennenswerte Auswirkung auf den Ablauf des Codes.

Geringe Auswirkung auf die Implementierung

In der Praxis findet man oft die `Runnable`-Variante, weil Java-Klassen in der Regel in eine Vererbungshierarchie eingebettet sind und daher selten von `Thread` erben können.

6.1.6 Threads am Programmende

Ein Java-Programm, das keine Threads erzeugt, endet regulär mit dem Ende von `main`. Allgemein gilt, dass ein Java-Programm regulär endet, wenn *alle Threads* beendet sind. Das folgende Beispielprogramm zeigt das:

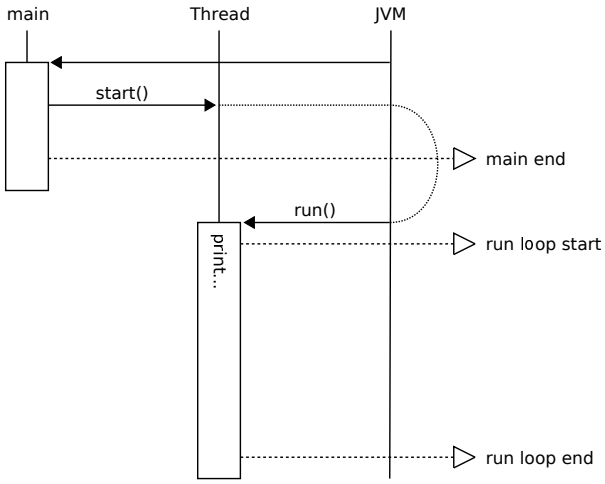
Programm endet mit dem letzten Thread

```
public class ThreadProgramEnd extends LoopPrinter {
    public static void main(String... args) {
        Thread thread = new ThreadProgramEnd();
        thread.start();
        System.out.println("main end");
    }
}
```

Listing 6.10: Programm endet als Ganzes, wenn alle Threads beendet sind.

Das Programm gibt aus:

```
$ ThreadProgramEnd
main end
run loop start
##########################################################################################
(und so weiter)
##########################################################################################
##########################################################################################run loop end
$
```



Daemon-Threads

Daemon-Threads verhindern Programmende nicht

Ein Thread kann mit einem Aufruf von

```
final void  setDaemon(boolean on)
```

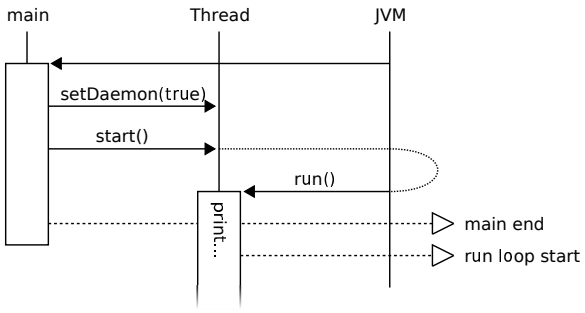
zu einem **Daemon-Thread** erklärt werden. Anders als normale Threads verhindern Daemon-Threads das Programmende nicht. Diese Eigenschaft muss allerdings *vor dem Start* des Threads festgelegt werden. Nach `start` führt der Aufruf von `setDaemon` zu einer `IllegalThreadStateException`.

Macht man im `ThreadProgramEnd` (Listing 6.10) den Thread `thread` zu einem Daemon:

```
Thread thread = new ThreadProgramEnd();
thread.setDaemon(true);
thread.start();
```

dann endet das Programm *ohne* Ausgabe von `run loop end`:

```
$ ThreadProgramEnd
main end
run loop start
############################
$
```



Der Thread `t` wird schon kurz nach dem Start wieder abgebrochen. Die Anzahl der ausgegebenen Hash-Zeichen schwankt, weil die Zeitverhältnisse in der JVM nicht genau vorhersagbar sind.

Daemon-Threads erledigen oft Verwaltungsaufgaben. Beispiele sind die Hintergrund-Threads, die die JVM automatisch startet (siehe Seite 397).

Aufgaben von Daemon-Threads

Methode `exit`

Die Methode `exit` der Klasse `Runtime`

```
void  exit(int exitcode)
```

Programm-abbruch unabhängig von Threads

beendet die JVM. Diese Methode kehrt nicht zurück. Die bequemere statische Methode der Klasse `System`

```
static void  exit(int exitcode)
```

ruft die `Runtime`-Methode indirekt auf.

Das `exit`-Argument signalisiert dem Betriebssystem, ob der ganze Programmlauf „erfolgreich“ war (Wert 0) oder nicht (ein anderer Wert als 0). Abgesehen von null und nicht null gibt es keine Übereinkunft bezüglich der Auslegung des „Exit-Codes“.[16] Auch für die Auslegung von „erfolgreich“ gibt es keine verbindliche Interpretation.[17]

`exit` stoppt alle Threads, ob Daemon oder nicht. Schiebt man im Programm `ThreadProgramEnd` (Listing 6.10) nach dem Konstruktoraufruf einen `exit`-Aufruf ein:

Keine Unterscheidung zwischen normalen und Daemon-Threads

[16] Auf Unix-Systemen (Linux, Mac OS X) zeigt ein positiver Exit-Code üblicherweise ein kleineres, überwindbares Hindernis an, ein negativer Code eine gravierende Fehlfunktion des Programms.

[17] Angenommen ein Programm durchsucht einen Datenbestand nach Anzeichen für Probleme. Ist das Programm „erfolgreich“, wenn es eine Anomalie findet oder wenn es keine findet?

```
Thread thread = new ThreadProgramEnd();
System.exit(0);
thread.start();
```

dann gibt das Programm überhaupt nichts aus. `System.exit` kehrt ja nicht zurück, deshalb wird der restliche Code von `main`, einschließlich `start` und der Ausgabe, nicht mehr ausgeführt.

6.2 Kommunikation mit Interrupts

In den bisher entwickelten Programmen laufen die Threads isoliert voneinander, das heißt ohne Kommunikation ab. Für einfache Anwendungen reicht das zwar aus, aber in der Regel werden sich Threads in irgendeiner Form abstimmen, das heißt, Informationen austauschen.

Interrupts als „Klopfzeichen“ zwischen Threads

Java stellt dazu verschiedene Möglichkeiten zur Verfügung. Eine sehr einfache Kommunikation erlauben **Interrupts**.[18]

6.2.1 Interrupt-Flags

Methoden zum Setzen und Prüfen des Interrupt-Flags

Interrupts beruhen auf „Interrupt-Flags“, die im Grunde nichts anderes als versteckte `boolean`-Variablen sind. Jedes `Thread`-Objekt verfügt über ein Interrupt-Flag, das zunächst gelöscht (`false`) ist. Die folgenden drei `Thread`-Methoden arbeiten mit dem Interrupt-Flag:

`void interrupt()`
: Setzt das Interrupt-Flag auf den Wert `true`. Wenn das Flag schon gesetzt war, hat der Aufruf keine Folgen.

`boolean isInterrupted()`
: Gibt Auskunft, ob das Interrupt-Flag gesetzt ist (`true`) oder nicht (`false`).

`static boolean interrupted()`
: Prüft, ob das Interrupt-Flag *des laufenden Threads* selbst gesetzt ist. Anders als der Getter `isInterrupted` löscht diese Methode das Flag nach dem Test. `interrupted` ist also *kein* Getter, weil es den Zustand des Objekts sowohl liest als auch ändert.

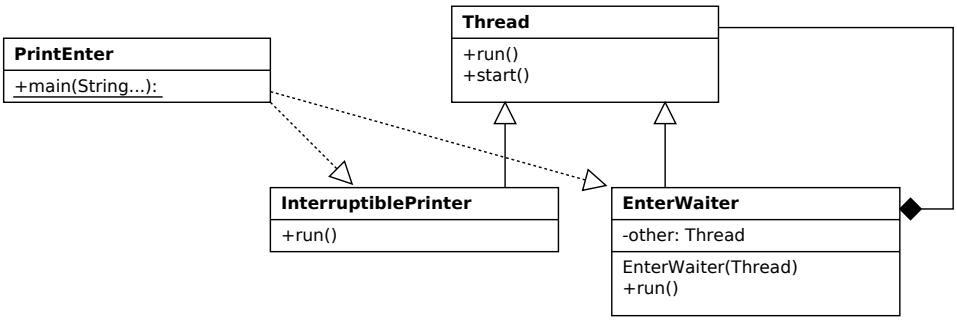
[18] Der Begriff „Interrupt“ taucht in der Informatik immer wieder auf. Zum Beispiel gibt es auf der Ebene des Betriebssystems Interrupts, ebenso wie in der Hardware. Hier geht es nur um „Java-Interrupts“. Diese verlassen ein Java-Programm nicht und haben nichts mit Interrupts außerhalb der JVM zu tun.

Diese Methode ist statisch. Sie richtet sich automatisch an den Thread des Aufrufers selbst, ohne dass dazu das entsprechende `Thread`-Objekt bekannt sein muss.

Freiwilliger Test des eigenen Interrupt-Flags

Ein Aufruf von `interrupt` setzt das Flag, hat aber zunächst weiter keine Folgen. Der `interrupt`-Aufrufer weiß nicht, ob der Empfänger-Thread das Flag überhaupt beachtet und wie er gegebenenfalls darauf reagiert. Die Kommunikation mit Interrupts beruht auf kooperierenden Threads.[19] Wenn ein Thread beschließen sollte sein Interrupt-Flag zu ignorieren, sind die anderen Threads machtlos.

Das folgende Programm soll so lange Zeichen ausgeben, bis der Benutzer die Eingabetaste drückt.[20] Es besteht aus drei Klassen:



- Die `Thread`-Klasse `InterruptiblePrinter` gibt in einer Schleife Zeichen aus, bis ein Interrupt eintrifft.

```
public class InterruptiblePrinter extends Thread {
    public void run() {
        while(!isInterrupted())
            System.out.print("#");
        System.out.println("got interrupt");
    }
}
```

Listing 6.11: Thread, der Zeichen ausgibt, bis ein Interrupt eintrifft.

- Die `Thread`-Klasse `EnterWaiter` wartet auf die Eingabetaste. Dann schickt es einen Interrupt an den Thread `other`.

[19] Man muss wohl nicht annehmen, dass sich die Threads in einem Programm feindlich gesonnen sind und untereinander bekriegen.

[20] Die Standardeingabe puffert Tastendrücke bis zur Eingabetaste. Das Programm reagiert also nur auf die diese Taste. Für den Zweck dieses Beispielprogramms ist diese Einschränkung unerheblich.

```
import java.io.*;

public class EnterWaiter extends Thread {
    private final Thread other;

    public EnterWaiter(Thread other) {
        this.other = other;
    }

    public void run() {
        System.console().readLine("Enter to send interrupt ... ");
        other.interrupt();
    }
}
```

Listing 6.12: Thread, der nach der Eingabetaste einen Interrupt an einen anderen Thread schickt.

Die `run`-Methode fängt die `IOException` von `read` auf und gibt sie als `RuntimeException` weiter. Das ist nötig, weil `run` keine Checked-Exception werfen darf.[21]

- Das Hauptprogramm `PrintEnter` startet je einen Thread der beiden vorhergehenden Klassen.

```
public class PrintEnter {
    public static void main(String... args) {
        Thread printer = new InterruptiblePrinter();
        Thread waiter = new EnterWaiter(printer);
        printer.start();
        waiter.start();
    }
}
```

Listing 6.13: Start von zwei Threads, von denen einer dem anderen einen Interrupt schickt.

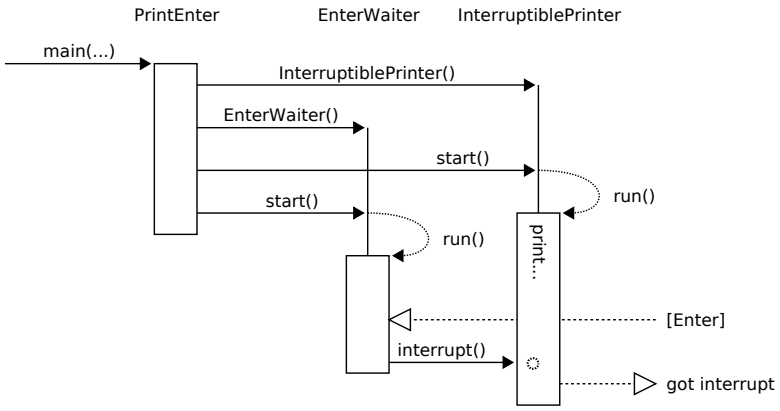
`main` endet nach dem Start der beiden Threads. Das Programm als Ganzes endet damit aber nicht, sondern läuft weiter, bis alle Threads beendet sind.

Nach dem Programmstart flutet die Ausgabeschleife des `InterruptiblePrinter` den Bildschirm. Ein Druck auf die Eingabetaste beendet den Spuk:

```
$ java PrintEnter
################################################################################
########################################Enter to send interrupt ... ############
################################################################################
```

[21] Ganz unproblematisch ist diese Umverpackung nicht, weil Checked-Exceptions (beispielsweise `IOException`) eine ganz andere Rolle als Unchecked-Exceptions (`RuntimeExceptions` und `Errors`) spielen. Für Erstere sind Gegenmaßnahmen denkbar, Letztere sind die Notbremse eines fehlerhaft geschriebenen Programms.

```
(und so weiter)
##########################################################################################
##########################################################################################
[Enter]
#[Enter]#
#######EnterWaiter.run: interrupt sent
got interrupt
$
```



6.2.2 `isInterrupted` und `interrupted`

Im einfachen Beispiel `InterruptiblePrinter` reicht der Getter `isInterrupted` aus, weil diese Klasse direkt von `Thread` erbt. Der Test

Test des Interrupt-Flags des gerade laufenden Threads

```
while(!isInterrupted())
```

im Schleifenkopf kann auch durch

```
while(!Thread.interrupted())
```

ersetzt werden. Er richtet sich an den Thread, der diese statische Methode `interrupted` aufgerufen hat. Das entsprechende `Thread`-Objekt muss dazu nicht bekannt sein. Selbst die Variante

```
while(!interrupted())
```

funktioniert in diesem Beispiel, weil sich die statische Methode auch über `this` aufrufen lässt.[22] Am Verhalten von `PrintEnter` (Listing 6.13) ändert sich nichts.

Statische `interrupted`-Methode in `Runnable`-Objekten

Nicht immer steht eine Referenz auf den laufenden Thread zur Verfügung. Ändert man beispielsweise die Klasse `InterruptiblePrinter` so ab, dass sie nicht mehr von `Thread` erbt, sondern `Runnable` implementiert, dann bleibt zum Test des Interrupt-Flags nur der Aufruf von `Thread.interrupted`. Die anderen beiden oben gezeigten Varianten scheiden aus:

```
public class InterruptibleRunnablePrinter implements Runnable {
    public void run() {
        while(!Thread.interrupted())
            System.out.print("#");
    }
}
```

Listing 6.14: Notwendigkeit der statischen Testmethode für Interrupts.

Diese Klasse *ist kein* `Thread` und kennt das `Thread`-Objekt auch nicht. Sie erbt `isInterrupted` nicht und kann diesen Getter nicht verwenden.[23]

6.2.3 `sleep` und `InterruptedException`

Reaktion länger blockierender Methoden auf Interrupts

Ein Thread muss aktiv das Interrupt-Flag überprüfen. Manche Methodenaufrufe blockieren allerdings für lange oder sogar unbestimmte Zeit, bevor sie zurückkehren. Ein Beispiel ist die statische `Thread`-Methode

```
static void  sleep(long millis)
```

die erst nach der angegebenen Anzahl Millisekunden zurückkehrt. `sleep` wartet passiv, das heißt, das Warten kostet keine Rechenleistung.

Wenn während eines längeren `sleep` ein Interrupt eintrifft, könnte ein Thread erst nach Ablauf der kompletten Wartezeit darauf reagieren, weil er das Interrupt-Flag vorher überhaupt nicht abfragen kann. Um das zu vermeiden, bricht `sleep` beim Eintreffen eines Interrupts sofort mit einer `InterruptedException` ab. Das Interrupt-Flag wird dabei gelöscht. Der Aufrufer von `sleep` kann sich frei entscheiden, ob er

[22] Der Aufruf einer statischen Methode über ein Objekt suggeriert einen normalen Methodenaufruf und ist daher für den Leser irreführend. Die Konstruktion ist in keinem Fall nötig und sollte besser vermieden werden. Dass Java sie überhaupt zulässt, hat historische Gründe.

[23] Die statische Methode `Thread.currentThread()` liefert das `Thread`-Objekt des gerade laufenden Threads. Daher könnte `Thread.currentThread().isInterrupted()` aufgerufen werden.

beispielsweise auf den Interrupt reagieren oder mit einem neuen Aufruf von `sleep` weiter warten möchte.

`sleep` kann also auf zwei Arten enden:

1. regulär nach Ablauf der Wartezeit mit normaler Rückkehr und
2. vorzeitig mit einer `InterruptedException`.

Die folgende Klasse `SlowInterruptiblePrinter` entspricht der Klasse `InterruptiblePrinter`, legt aber nach jedem Zeichen eine Pause von einigen Sekunden ein. Drückt nun der Benutzer die Eingabetaste, so trifft ein Interrupt ein und unterbricht den `sleep`-Aufruf. Das führt zu einer `InterruptedException`, die direkt nach dem `sleep` aufgefangen wird.

```
public class SlowInterruptiblePrinter extends Thread {
    public void run() {
        while(!Thread.interrupted()) {
            out.print("#");
            try {
                Thread.sleep(5000);
            }
            catch(InterruptedException ex) {
            }
        }
        out.println("got interrupt");
    }

}
```

Listing 6.15: `InterruptedException` aus `sleep` beim Eintreffen eines Interrupts.

In dieser Form funktioniert das Programm noch nicht, weil die `InterruptedException` das Interrupt-Flag löscht. Die Schleifenbedingung findet also ein gelöschtes Interrupt-Flag vor und die Schleife läuft nach der Exception einfach weiter. Ein Aufruf von `interrupt` an das *eigene* `Thread`-Objekt setzt das eben gelöschte Flag im `catch`-Block erneut und stoppt damit auch die Schleife:

Interrupt-Flag nach `InterruptedException` immer zurückgesetzt

```
while(!Thread.interrupted()) {
    out.print("#");
    try {
        Thread.sleep(5000);
    }
    catch(InterruptedException ex) {
        interrupt();
    }
}
```

Listing 6.16: Selbst-Interrupt für ein geordnetes Ende der Schleife.

Das Programm `SlowPrintEnter` ist identisch mit `PrintEnter` (Listing 6.13), arbeitet aber mit einem `SlowInterruptiblePrinter` statt eines `InterruptiblePrinter` (Listing 6.11). Jetzt beendet die Eingabetaste auch die langsame Ausgabe augenblicklich:

```
$ java SlowPrintEnter
[Enter]
#[Enter]#
got interrupt
$
```

Das Neu-Setzen des Interrupt-Flags ist nicht mehr nötig, wenn die komplette Schleife in den `try`-Block verschoben wird. Eine `InterruptedException` beendet `run` sofort. Dass das Interrupt-Flag gelöscht ist, spielt dabei keine Rolle mehr.

```
try {
    while(!Thread.interrupted()) {
        out.print("#");
        Thread.sleep(5000);
    }
}
catch(InterruptedException ex) {
}
```

Listing 6.17: Selbst-Interrupt unnötig beim Fangen der `InterruptedException` außerhalb der Schleife.

Komplett weglassen kann man den Test des Interrupt-Flags dennoch nicht, wie im folgenden fehlerhaften Code:

```
try {
    while(true) {
        out.print("#");
        Thread.sleep(5000);
    }
}
catch(InterruptedException ex) {
}
```

Listing 6.18: Potenzieller Fehler beim Wegfall des Tests auf Interrupt.

Trifft der Interrupt während des `sleep`-Aufrufs ein, dann endet die Schleife planmäßig. Trifft er aber ein, während der Rest des Schleifencodes läuft, dann bleibt er unerkannt![24]

[24] Im Vergleich zum `sleep` braucht der Rest des Schleifencodes wenig Zeit. Die Wahrscheinlichkeit, dass der Interrupt während des `sleep`-Aufrufs eintrifft, ist deshalb groß, aber nicht 100%.

Die Methoden `wait` und `join` können auf unbestimmte Zeit blockieren. Sie brechen daher wie `sleep` bei einem Interrupt mit einer `InterruptedException` ab, um dem Aufrufer die Gelegenheit zu geben, auf den Interrupt nach seinen Vorstellungen zu reagieren.

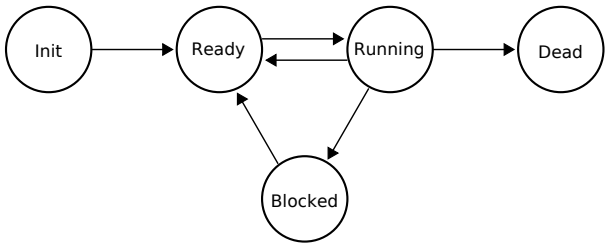
`InterruptedException` bei `wait` und `join`

6.3 Scheduling und Multiprozessoren

6.3.1 Lebenslauf eines Threads

Ein Thread kann genau einmal ablaufen. Das `Thread`-Objekt existiert allerdings schon vor dem Start und auch noch nach dem Ende. Es durchläuft im Laufe seines „Daseins“ verschiedene Zustände, deren Abfolge als Lebenszyklus (*life cycle*) bezeichnet wird. Zu jedem Zeitpunkt befindet sich jeder Thread in genau einem dieser Zustände.[25] Das folgende Diagramm zeigt diesen Lebenszyklus im Überblick:[26]

Zustandsmodell aus Betriebssystemen übernommen



Die folgende Liste beschreibt die Idee hinter den Zuständen. Etwas komplizierter sind die Umstände der *Übergänge* zwischen den Zuständen. Darauf geht der nächste Abschnitt ein.

Merkmale der Zustände

Init (auch *New*, *Start*)
: Ein neu erzeugtes `Thread`-Objekt besteht lediglich aus Daten auf dem Heap. In der JVM sind noch keine Ressourcen für den Ablauf des Threads zugewiesen. Das `Thread`-Objekt kann benutzt werden wie jedes andere Objekt.

[25] Das gleiche Modell beschreibt den Lebenszyklus von Prozessen in Betriebssystemen. Es hat sich dort lange vor Java bewährt und wurde deshalb für Java-Threads übernommen.

[26] Andere Texte verwenden zum Teil andere Namen für die Zustände. Das ändert nichts am grundsätzlichen Aufbau und an der Interpretation des Modells.

Ready (*Runnable*, *Ready-to-run*)
: Mit dem Aufruf von `start` arrangiert die JVM die Verwaltungsdaten eines neuen Threads. Sobald alles bereit ist, reiht sich der neue Thread in die Menge aller Bewerber ein, die sofort losrechnen könnten, wenn nur ein Prozessor[27] frei wäre. *Ready* ist das Wartezimmer der Prozessoren. Dort drängeln sich all die Aspiranten, die versuchen, einen Prozessor für sich zu ergattern.

Running
: Der Thread beansprucht einen Prozessor für sich und führt Code aus. Das beginnt mit dem Aufruf der `run`-Methode und setzt sich durch deren Rumpf fort. Der Thread verbraucht hier Rechenleistung. *Running* ist der Traum jedes Threads, das Ziel seines Daseins.

Dead (*Terminated*)
: Mit dem Ende der `run`-Methode wechselt der Thread in den Zustand *Dead*. Hier bleibt er so lange, bis die JVM das Objekt freigibt. Bis dahin kann das Objekt weiterhin benutzt werden, wie jedes andere Java-Objekt. Aus dem Zustand *Dead* gibt es keine Rückkehr.

Blocked (*Not-runnable*, *Waiting*)
: In diesem Zustand wartet ein Thread darauf, dass eine bestimmte Voraussetzung geschaffen wird, ohne die er nicht weiterrechnen kann oder darf. Das kann zum Beispiel eine Eingabe sein, der Ablauf einer Zeitspanne oder die Freigabe eines Monitors (mehr über Monitore finden Sie in Abschnitt 6.4).

  Anders als im Zustand *Ready* könnte ein Thread im Zustand *Blocked* auch dann nicht fortfahren, wenn ein Prozessor frei wäre. Darüber hinaus kann der Thread die Voraussetzung nicht aus eigener Kraft herbeiführen, sondern hängt von (aus seiner Sicht) äußeren Einflüssen ab.

▶ In `java.lang` ist der Aufzählungstyp `Thread.State` für Thread-Zustände definiert, die beispielsweise der `Thread`-Getter `getState()` liefert. Die Elemente von `Thread.State` weichen etwas vom hier gezeigten Modell ab. So gibt es in `Thread.State` mehrere verschiedene blockierende Zustände, ein Zustand *Running* fehlt ganz und Threads in I/O-Operationen werden anders behandelt. Die folgende Tabelle stellt die hier verwendeten Bezeichnungen den Aufzählungselementen gegenüber:

[27] Der Begriff „Prozessor“ bezeichnet eine CPU oder einen CPU-Core (siehe Seite 393).

Init	`NEW`
Ready	`RUNNABLE`
Running	-
Dead	`TERMINATED`
Blocked	`BLOCKED`, `WAITING`, `TIMED_WAITING`

Das Zustandsmodell in diesem Text ist kein exaktes Abbild von `Thread.State`, dafür aber einfacher. ◀

6.3.2 Scheduler

Oberaufsicht über alle Threads

Natürlich darf nicht jeder Thread nach eigenem Belieben seinen Zustand wechseln. Es muss eine koordinierende Instanz geben, die für ein geordnetes Miteinander sorgt. Diese Instanz ist der **Thread-Scheduler**, technisch gesehen ein Algorithmus in der JVM.[28] Der Scheduler verwaltet alle Threads und steuert deren Zustände und Zustandsübergänge.[29]

Mögliche Pfade durch den Zustandsgraphen

Das oben gezeigte Zustandsdiagramm lässt nur bestimmte Übergänge zu. So gibt es beispielsweise keinen direkten Weg von *Blocked* nach *Running* und keinen Ausgang aus *Dead*. Grundsätzlich wechselt ein Thread einmal von *Init* in den mittleren Teil und später einmal aus dem mittleren Teil nach *Dead*. Im mittleren Teil kreist der Thread zwischen *Ready* und *Running*, möglicherweise auf dem Umweg über *Blocked*.

Automatisch und explizit gesteuerte Übergänge

Einige Übergänge können aktiv ausgelöst werden, der Rest unterliegt der Kontrolle des Schedulers und anderen Einflüssen. Die folgende Liste beschreibt die Übergänge:

Init → *Ready*
: Diesen Übergang löst der Aufruf von `start` aus. Der Scheduler lässt einen Thread aber nicht sofort im Zustand *Running* „loslegen“, sondern reiht ihn zunächst in die Bewerber im Zustand *Ready* ein.

[28] Auch in Betriebssystemen gibt es Scheduler, der die gleiche Aufgabe für die Prozesse des Betriebssystems wahrnehmen. Der Scheduler der JVM ist nur für die Threads *innerhalb* einer JVM zuständig. Er kann sich mit einem Scheduler des Betriebssystems arrangieren, muss das aber nicht tun.

[29] Manche Texte gebrauchen den Begriff „Scheduler“ enger für einen Algorithmus, der sich nur um regelmäßig wiederkehrende Abläufe kümmert.

Ready → *Running*
: Nach Kriterien, die unten diskutiert werden, wählt der Scheduler aus all den Bewerbern im Zustand *Ready* einen Kandidaten aus, dem er einen Prozessor zuteilt. Dieser Thread wird in den Zustand *Running* „erhoben“.

Running → *Dead*
: Mit dem Ende von `run` endet der Thread. Er wechselt dann in den Zustand *Dead*, aus dem es keine Rückkehr gibt.

Running → *Ready*
: Der Scheduler sollte[30] darauf achten, dass nicht ein einziger egoistischer Thread den (vielleicht einzigen) Prozessor für sich belegt und zu keinem Ende kommt, während die übrigen Bewerber im Zustand *Ready* versauern. Deshalb wird der Scheduler einen rechnenden Thread bei Gelegenheit unterbrechen und in den Zustand *Ready* zurückschicken. Ob und wann genau das geschieht, ist Sache des Schedulers (siehe nächster Abschnitt).
Ein Thread kann diesen Wechsel nicht verhindern und auch nicht den Zeitpunkt bestimmen. Der Thread wird dabei nicht beendet, sondern bloß vorübergehend zu einer Pause gezwungen. Wenn er später wieder an die Reihe kommt, findet er exakt die gleiche Umgebung vor, aus der er verdrängt wurde. Aus logischer Sicht kann ein Thread die Wechsel zwischen *Running* und *Ready* nicht erkennen.[31]

Running → *Blocked*
: Manche Methoden sind darauf angewiesen, dass bestimmte Voraussetzungen erfüllt sind. Der Scheduler erkennt das und schiebt einen Thread, der eine „blockierende Methode“ aufruft, in den Zustand *Blocked* ab. Der Thread ist vorerst „aus dem Rennen“ und bis auf Weiteres vom Wettbewerb um die Prozessoren ausgeschlossen.
Beispiele blockierender Methoden sind

  - `read`, `write` und fast alle anderen I/O-Methoden, die auf Eingaben oder auf die Zustellung von Ausgaben warten,
  - `sleep`, das den Ablauf einer vorgegebenen Zeitspanne abwartet,
  - `wait`, das die Freigabe eines Monitors erfordert,
  - `join`, das auf das Ende eines anderen Threads wartet.

  Im Gegensatz dazu sind beispielsweise `Math.sin` oder `Arrays.sort` keine blockierenden Methoden.

[30] Die Betonung liegt auf *sollte*. Er ist dazu nicht verpflichtet, wie im nächsten Abschnitt ausgeführt wird.

[31] Vielleicht könnte eine Beobachtung der Systemzeit Rückschlüsse erlauben. In der Regel arbeiten die verfügbaren Zeitquellen aber nicht genau genug, um den Threadwechsel zuverlässig zu bestimmen.

Blocked → *Running*
: Sobald die Bedingung erfüllt ist, auf die ein Thread im Zustand *Blocked* wartet, könnte der Thread eigentlich fortfahren. Er darf aber nicht sofort wieder einen Prozessor in Beschlag nehmen, sondern muss sich zuerst unter die übrigen Bewerber im Zustand *Ready* einreihen und kann dort auf sein Glück hoffen.

6.3.3 Expliziter Zustandswechsel

`sleep` wartet passiv eine Zeitspanne ab

Die statische Methode `sleep` legt einen Thread für eine bestimmte Anzahl Millisekunden schlafen (siehe Seite 384):

```
static void  sleep(long millis) throws InterruptedException
```

Ein Thread wartet in dieser Zeit im Zustand *Blocked* und verbraucht dabei keine Rechenleistung. Besonders bei kleinen Argumenten ist `sleep` allerdings nicht sehr zuverlässig. Das folgende Programm gibt in zwei parallel laufenden Schleifen jeweils 1000 Zeichen aus. Nach jedem Zeichen wartet das Programm die Anzahl Millisekunden, die auf der Kommandozeile festgelegt ist:

```
import static java.lang.System.*;

public class LoopSleepPrinter extends Thread {
    private static int millis;

    public void run() {
        try {
            out.println("run loop start");
            long start = currentTimeMillis();
            for(int i = 0; i < 1000; i++) {
                out.print("#");
                Thread.sleep(millis);
            }
            out.printf("run loop end, %d millis%n", currentTimeMillis() - start);
        }
        catch(InterruptedException ex) {
            throw new RuntimeException(ex);
        }
    }

    public static void main(String... args) throws InterruptedException {
        millis = Integer.parseInt(args[0]);
        new LoopSleepPrinter().start();
        out.println("main loop start");
        long start = currentTimeMillis();
        for(int i = 0; i < 1000; i++) {
            out.print(":");
            Thread.sleep(millis);
```

```
        }
        out.printf("main loop end, %d millis%n", currentTimeMillis() - start);
    }
}
```

Listing 6.19: Ausgabe von Zeichen in zwei Threads mit kurzer Verzögerung.

Startet man das Programm mit dem Argument 1, dann warten die Schleifen nach jedem Zeichen 1 Millisekunde lang. Die beiden Schleifen sollten jeweils etwa $1000 \cdot 1\text{ms} = 1$ Sekunde lang laufen. Tatsächlich braucht das Programm aber viel länger:

```
$ java LoopSleepPrinter 1
main loop start
:run loop start
#:#::#:#:#:#:#:#:#::::#:#:#::#:#::#::#::#::#::#::::#:#:::#:#:#::#::#::#:#:#::#:#:
:#::#::#::#::#:##::##::#:#:##::#:#:#:##:#::##::##::#:#:##::#:#:#:##:#::##::#:#:##::#
:#:#:##:#::##::#:#:#:##:#::##:#::##::##::#:#:##::#::#::#::#::##:#:#:#:#:#:#::#:##:#:
(und so weiter)
:::::#:#:#:#:#:::#:#:#:#:::::#:#:::#::#::#::#:::#:#:#:#:#:#:#::#::#::##:#:#:#:#:#:##::
#:#:#:#:##::##::#:#:##::#::#::#::#::##:#:#:#:#:#:#::#:##:##:#:#:#:#:#:##::#:#:#:#:##
::##::#:#:##::#::#::#::#::##:#:#:#:#:#:#::#:##:##::#:#:#:#:#:##::##::#:#:##::#::#::#::
#:##
run loop end, 4412 millis
::::main loop end, 4456 millis
$
```

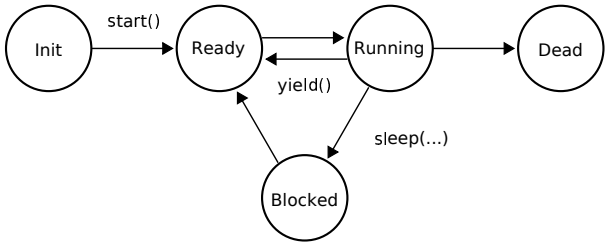
Die Zeichen erscheinen paarweise auf dem Bildschirm, weil die Wartezeit von einer Millisekunde der JVM ausreichend Gelegenheit gibt, den jeweils anderen Thread in den Zustand *Running* zu verschieben, wo er sein Zeichen ausgibt. Die Reihenfolge der Zeichen innerhalb eines Paars wechselt aber gelegentlich, weil sich die Schleifen minimal gegeneinander verschieben.

`yield` gibt den Prozessor freiwillig auf

Ein Thread kann den Prozessor freiwillig mit dem Aufruf der Methode

```
static void  yield()
```

aufgeben. Er kehrt damit direkt in den Zustand *Ready* zurück, ohne den Weg über *Blocked* zu nehmen.



Tauscht man in `LoopSleepPrinter` (Listing 6.19) den `sleep`-Aufruf gegen `yield` aus, dann lässt sich die Arbeit des Schedulers gut erkennen. Ausnahmsweise ist hier die gesamte Ausgabe ungekürzt abgedruckt:

```
$ java LoopYieldPrinter
main loop start
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::run loop start
#:::::#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::############################################
##########################################::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::#:::::::::::::::::::::::::::::::#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:##:#:
#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:##:#:#
:#:#:#:#:################:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:
#:##:#:#:#:#:#:#:#:#:#:#:##:#:#:#:#:#:#:#:#:#:#:#:#:#::#:#:#:#:#:#:#:#:#:#:#:#:#:#:#
:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#
:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#::#:#:#:#:#:#:#:#:
#:#:#:#:#:#:#:#::#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#
:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:##:#:#:#:#:#:#:#:#:#:#:#:#:#::#:#:#:#
:#:#:#:#:#:#:#:#:#:#:#:#:#::###############:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:
#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:
#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:
#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:
#:#:#:#:#:##:#:#:#:#:#:#:#:#:#:#:#:#:#############:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#
:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#::##:#:#:#::#######
####################################################################################
####################################################################################
####################################################################################
###################:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#:#::#:#:#:#:#
:#:#:##:#:#:#:#:#:#:#::::::::::::::::::run loop end, 47 millis
::::::::::::::::::::::main loop end, 66 millis
```

`yield` kann nicht garantieren, dass ein *anderer* Thread an die Reihe kommt. Der Scheduler kann aus den Kandidaten gerade den Thread wieder auswählen, der eben `yield` aufgerufen hat. Der Nutzen von `yield` hält sich in Grenzen. Die Methode lässt sich aber durchaus einsetzen, um einem gleichmäßigen Fortschritt von Threads zu begünstigen.

6.3.4 Threads und Prozessoren

CPUs und Cores

Java verteilt Threads auf Prozessoren

Aktuelle Rechner sind in der Regel mit mehreren CPUs beziehungsweise CPU-Cores ausgestattet, die im folgenden als „Prozessoren“ bezeichnet werden. Mehrfache Prozessoren sollten möglichst gut genutzt werden:

1. Java sollte Threads auf möglichst viele Prozessoren verteilen.
2. Bei einer Überzahl von Threads sollte die verfügbare Rechenleistung gleichmäßig aufgeteilt werden.

Die Anforderungen sind unabhängig voneinander. Für die erste ist Java auf die Unterstützung des Betriebssystems angewiesen. Die zweite Forderung ist Sache des Schedulers innerhalb der JVM.

Time-Slicing

Die Java-Standard-Edition von Oracle (Oracle Java SE 7) kommt beiden Forderungen nach.

1. Die JVM arbeitet mit den populären Desktop-Betriebssystemen zusammen (Windows, Linux, Mac OS X) und nutzt jeweils alle Prozessoren. Deren Anzahl liefert die Methode `availableProcessors` der Klasse `Runtime`, wie im folgenden Beispielprogramm:

```
public class PrintProcessors {
    public static void main(String... args) {
        Runtime runtime = Runtime.getRuntime();
        System.out.println(runtime.availableProcessors());
    }
}
```

Listing 6.20: Ausgabe der Anzahl Prozessoren.

Mehr Threads als Prozessoren: Threads wechseln sich ab

2. Ein Java-Programm kann mehr Threads starten als Prozessoren zur Verfügung stehen.[32] Der Scheduler sorgt in diesem Fall für Gerechtigkeit und lässt die Threads abwechselnd rechnen. Jeder Thread erhält den Prozessor für eine gewisse Zeit und muss dann wieder Platz machen für den nächsten Thread. Die gesamte Prozessorzeit wird in „Zeitscheiben“ zerschnitten, die den Threads reihum zugewiesen werden. Dieses Verfahren wird deshalb als Time-Slicing- oder Round-Robin-Strategie bezeichnet.

Threadwechsel transparent

Ein einzelner Thread nimmt das Time-Slicing nicht direkt wahr. Aus seiner Sicht läuft das System einfach etwas langsamer, als die Hardware erwarten lässt.
Die Länge der Zeitscheiben bestimmt die Frequenz der Threadwechsel.

- Je kürzer die Zeitscheiben ausfallen, desto gleichmäßiger stellt sich der Ablauf für jeden einzelnen Thread dar.
- Je länger die Zeitscheiben ausfallen, desto weniger fällt der Verwaltungsaufwand für die Threadwechsel ins Gewicht.

[32] 1000 Threads sind heute auch für vergleichsweise leistungsschwache Netbooks kein Problem.

Der Scheduling-Algorithmus sucht einen Kompromiss zwischen diesen sich widersprechenden Anforderungen.[33]

So selbstverständlich die beiden oben genannten Anforderungen auch klingen mögen, eine konkrete Implementierung von Java *muss* keine davon erfüllen.

JVM möglicherweise ohne Time-Slicing

- Eine JVM kann mehrfache Prozessoren ignorieren und nur einen einzigen zur Kenntnis nehmen.
- Der Scheduler kann Time-Slicing ignorieren und einen einzigen Thread exklusiv rechnen lassen, während die anderen Bewerber für immer im Zustand *Ready* „verhungern".

Ein portables Java-Programm mit mehreren Threads muss auch unter solch widrigen Umständen noch funktionieren und kann nicht stillschweigend unterstellen, dass es mehrere Prozessoren gibt oder dass der Scheduler Time-Slicing umsetzt.

Während Java auf den Desktop-Systemen oder noch leistungsfähigerer Hardware den Anforderungen problemlos nachkommen kann, gilt das für kleine und leistungsschwache Systeme (*embedded system*) nicht unbedingt. Dort stehen möglicherweise nur so knappe Ressourcen zur Verfügung, dass ein Scheduler zu kostspielig ist.

Performance-Beispiel

Das folgende Beispielprogramm startet mehrere Threads, von denen jeder einzelne getrennt einige Fibonaccizahlen berechnet (siehe Seite 256).[34] Die Anzahl der Threads wird auf der Kommandozeile festgesetzt. Alle Threads rechnen hier ohne Unterbrechung. Es gibt praktisch keine Pausen durch Aufrufe blockierender Methoden, abgesehen von den Ausgaben, die aber im Vergleich zur restlichen Rechenzeit nicht ins Gewicht fallen. Das Programm `Fibonacci` belastet weder den Heap noch den Stack nennenswert, daher ist auch keine Garbage-Collection nötig.

Mehrere Prozessoren arbeiten gleichzeitig

```
import java.util.*;

public class ParallelFibonacci extends Thread {
    public static void main(String... args) throws InterruptedException {
```

[33] Zeitscheiben von Oracle Java SE 7 auf Linux sind etwa 3–4 Millisekunden lang. Die konkrete Länge hängt von der einzelnen Java-Implementierung ab.

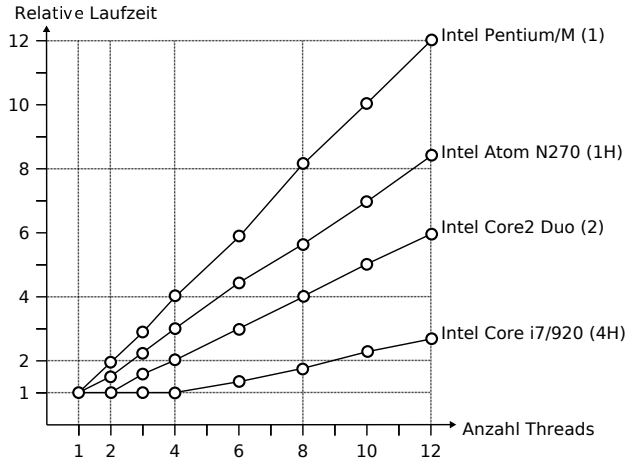
[34] Das Argument der `main`-Methode der Klasse `Fibonacci` ist nur der String `"40"` und kein Array von Strings. Das funktioniert, weil `main` mit einem Vararg-Parameter definiert ist und nicht mit einem String-Array. Für den Aufruf von `main` durch die JVM beim Programmstart ist das ohne Bedeutung.

```
        int num = Integer.parseInt(args[0]);
        List<Thread> threadList = new ArrayList<>();
        for(int t = 0; t < num; t++)
            threadList.add(new Thread() {
            public void run() {
                Fibonacci.main("40");
            }
        });
        long start = System.currentTimeMillis();
        for(Thread thread: threadList)
            thread.start();
        for(Thread thread: threadList)
            thread.join();
        System.out.printf("%d threads, %d millis%n", num, System.currentTimeMillis() - st
    }
}
```

Listing 6.21: Dieselbe aufwendige Berechnung gleichzeitig in mehreren Threads.

Beschleunigung bei verschiedenen CPU-Modellen

Das folgende Diagramm zeigt, wie lang das Programm auf verschiedenen CPU-Modellen braucht, um eine bestimmte Anzahl gleicher Threads auszuführen.[35] Hinter dem CPU-Modell ist die Anzahl der physischen Cores angegeben. Der Zusatz „H“ steht für Intels Hyperthreading-Technologie, die einen Core wie zwei „virtuelle“ Cores erscheinen lässt.



Dieser Test des Programms auf verschiedenen CPUs mit Oracle Java 7 SE (Linux) zeigt, dass diese Java-Implementierung die beiden Anforderungen erfüllt, also alle verfügbaren Prozessoren nutzt und Threads gleichmäßig darauf verteilt.

Gleichmäßige Auslastung der Prozessoren

Die Gesamtlaufzeit von `ParallelFibonacci` (Listing 6.21) bleibt gleich, bis die An-

[35] Die „relative Laufzeit“ bezieht sich auf die Laufzeit des Programms mit einem Thread. Jeder Programmaufruf wurde fünfmal wiederholt, davon das größte und das kleinste Ergebnis verworfen und die anderen drei gemittelt.

zahl der Threads die Anzahl der „echten“ Cores erreicht. Ab diesem Punkt folgen die Ergebnisse bei CPUs ohne Hyperthreading ungefähr einer Geraden mit der Steigung $\frac{1}{Cores}$.[36]

Diese Zahlen zeigen auch, dass ein ansonsten unbelastetes System optimal ausgelastet ist, wenn gerade so viele rechenintensive Threads laufen, wie Prozessoren zur Verfügung stehen. Mehr Threads zu starten bringt keinen zusätzlichen Nutzen, weniger Threads lassen Rechenleistung brach liegen. Obwohl die JVM recht effizient mit Threads umgeht, macht sich bei einer steigenden Anzahl von Threads auch irgendwann der schiere Verwaltungsaufwand bemerkbar und schmälert den Nutzen weiter.

6.3.5 Hintergrund-Threads

`jconsole` zur Beobachtung eines laufenden Programms

Die virtuelle Maschine ruft beim Programmstart die Methode `main` in einem neuen Thread auf. Gleichzeitig läuft noch eine Anzahl weiterer Threads, die Verwaltungsaufgaben der JVM wahrnehmen. Diese Threads sind selten aktiv und kosten in der Regel kaum Rechenleistung. Eine Ausnahme ist der Garbage-Collector, der einigen Aufwand erfordern kann.

Das Werkzeug `jconsole`, das mit dem JDK installiert wird, kann eine bereits laufende JVM überwachen und dabei die Hintergrund-Threads sichtbar machen. Als Beispiel dient das folgende Programm:

```
public class Sleep4Ever {
    public static void main(String... args) throws InterruptedException {
        Thread.sleep(Integer.MAX_VALUE);
    }
}
```

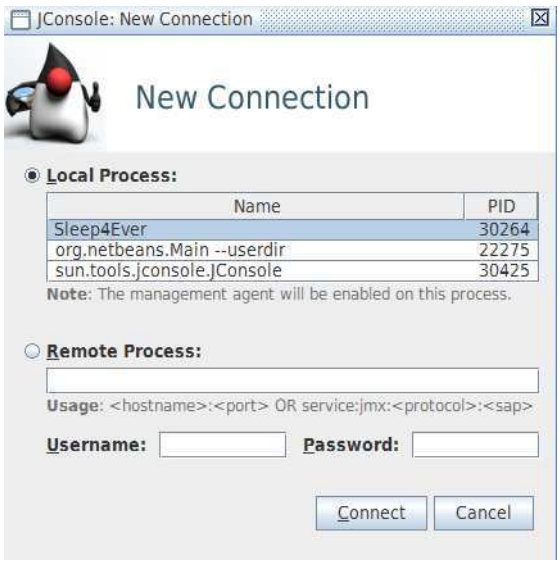
Listing 6.22: Testprogramm zur Untersuchung mit dem Werkzeug `jconsole`.

Es blockiert nach dem Start praktisch für immer

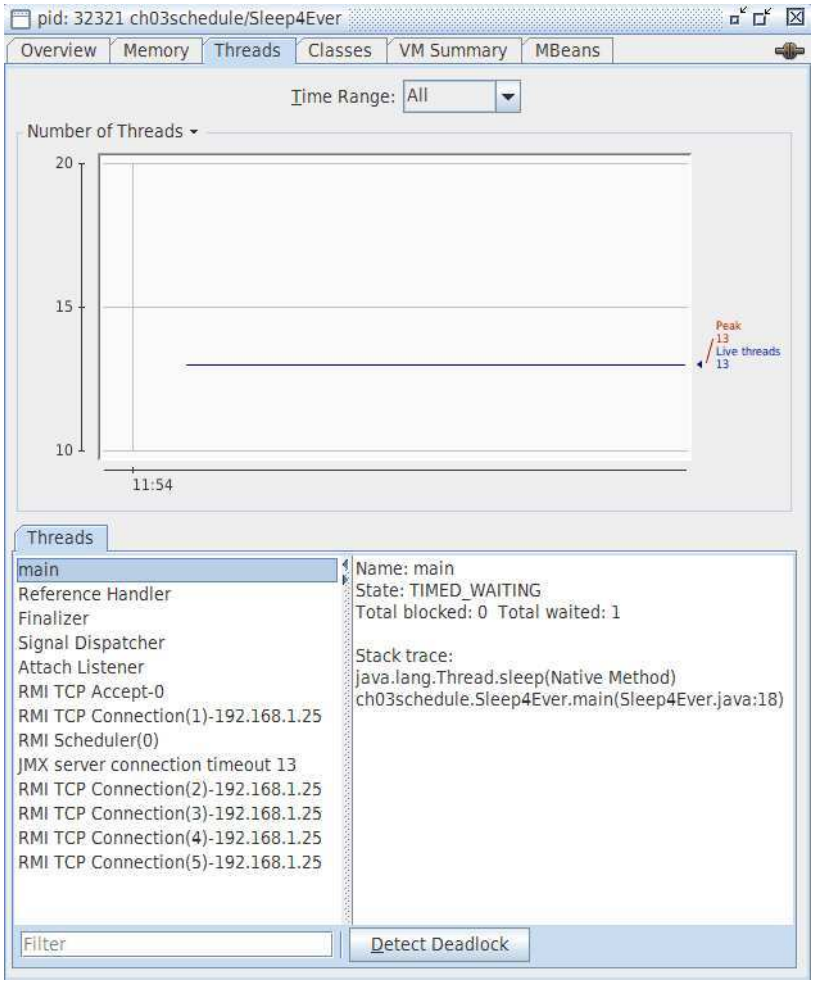
```
$ java Sleep4Ever
```

Wenn man in einer anderen Eingabeaufforderung `jconsole` startet, dann bietet das Werkzeug zuerst in einem Pop-up-Fenster alle momentan laufenden JVMs an:

[36] Aus den Ergebnissen lässt sich bezüglich Hyperthreading schließen, dass ein „virtueller“ Core in dieser Anwendung knapp halb so viel wie ein „echter“ Core leistet.



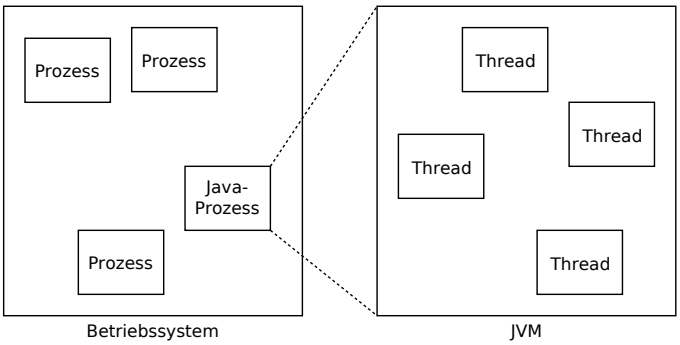
Nach Auswahl von `Sleep4Ever` öffnet sich ein Übersichtsfenster mit Informationen zur betreffenden JVM. Der Reiter „Threads“ blendet weitere Angaben zu den Threads ein. Im folgenden Beispiel gibt es 13 Threads, darunter den `main`-Thread. Die konkrete Ausgabe hängt vom einzelnen System ab.



6.3.6 Threads und Prozesse

Java-Threads und Prozesse im Betriebssystem

Ein Betriebssystem verwaltet laufende Programme als „Prozesse“. Auch ein Java-Programm ist aus der Sicht des Betriebssystems ein Prozess. Das Java-Programm kann seinerseits mehrere Threads starten, die innerhalb der JVM laufen.



Eine JVM kann die Unterstützung des Betriebssystems in Anspruch nehmen und die eigenen Threads auf dessen Ressourcen abbilden. Im Allgemeinen kann man aber nicht davon ausgehen, dass Java-Threads für das Betriebssystem sichtbar sind.

6.4 Konkurrierender Zugriff und Synchronisation

Zugriff auf gemeinsame Datenstrukturen

Threads erlauben effizientere Programme mit lang laufenden, blockierenden Operationen (typischerweise I/O), weil sich Pausen für andere Arbeiten nutzen lassen. Dabei reicht schon ein einzelner Prozessor aus. Mehrfache Prozessoren können auch Programme beschleunigen, die keine blockierenden Operationen verwenden, weil sich Teilaufgaben gleichzeitig erledigen lassen.

In aller Regel müssen Threads kommunizieren, beispielsweise um Teilergebnisse zu einem Gesamtergebnis zu kombinieren. Eine einfache, aber auch beschränkte Möglichkeit sind Interrupts (siehe Abschnitt 6.2). Mit gemeinsamen Datenstrukturen lassen sich mehr Informationen zwischen Threads austauschen. So naheliegend diese Idee ist, so diffizil sind die praktischen Probleme.

6.4.1 Thread-lokale und geteilte Daten

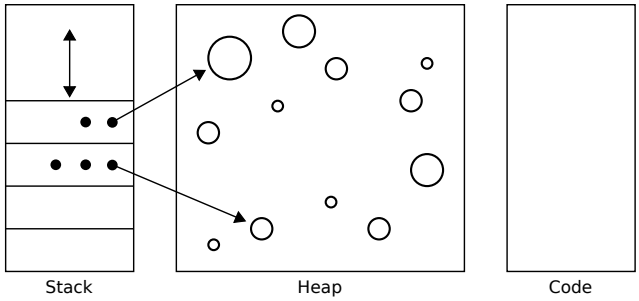
Heap für alle Threads, Stacks getrennt

Die wesentlichen Speicherbereiche eines Java-Programms sind Code, Stack und Heap (siehe Abschnitt 4.5.3, Seite 286).

- Der **Code** enthält den Bytecode und dazu noch konstante Daten, wie String-Literale und andere Konstanten, die dem Compiler schon bekannt sind. Dieser Bereich ist unveränderlich (*read only*).

- Der **Heap** nimmt Objekte auf, unabhängig davon, wie sie geschaffen werden.[37]
- Auf dem **Stack** liegen die lokale Variablen und Parameter.[38]

Die folgende Skizze veranschaulicht diese Speicherbereiche. Lokale Variablen beziehungsweise Parameter sind als schwarze Punkte auf dem Stack dargestellt. Referenzen zwischen Objekten auf dem Heap sind weggelassen, ebenso wie konstante Daten im Code.



Jeder Thread verfügt über seinen eigenen Stack, der von den Stacks anderer Threads vollkommen isoliert ist. Das bedeutet, dass die lokalen Variablen und Parameter einem Thread alleine gehören.

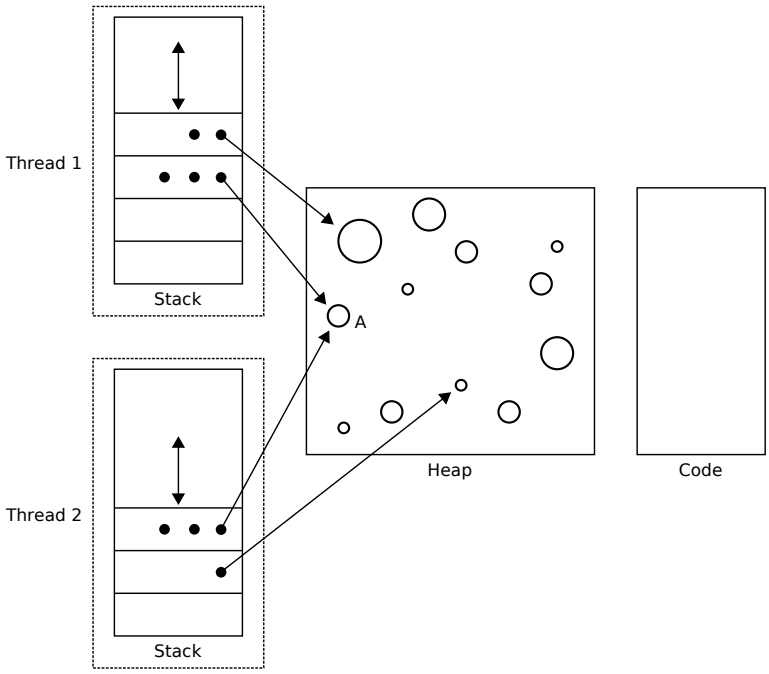
Informationsaustausch über gemeinsame Objekte

Alle Threads teilen sich dagegen Code und Heap. Ersterer ist ohnedies unveränderlich und daher unkritisch. Der geteilte Heap eröffnet einerseits die Möglichkeit zum Datenaustausch, macht andererseits aber auch einige Vorsichtsmaßnahmen nötig, die im Rest dieses Abschnitts erarbeitet werden.

Die folgende Skizze zeigt die Verhältnisse in einem Programm mit zwei Threads:

[37] Die meisten Objekte entstehen durch Konstruktoraufrufe, andere durch Cloning oder Deserialisierung. Konstruktoraufrufe können mit explizitem `new` ausgelöst werden oder implizit, wie beispielsweise bei Array-Literalen.

[38] Dazu kommen weitere verborgene Verwaltungsdaten von Methodenaufrufen, wie zum Beispiel Rückkehradressen, Methodennamen und Zeilennummern. Der Stack ist in Stackframes organisiert.



In diesem Bild referenzieren beide Threads dasselbe Objekt A. Daten, die Thread 1 in A einträgt, kann Thread 2 wieder herauslesen. Auf diese Art können die beiden Threads in Verbindung treten. Dabei sind grundsätzlich zwei Probleme zu lösen:

Probleme mit geteilten Objekten

1. Es darf nicht passieren, dass Thread 2 aus dem Objekt liest, *während* Thread 1 noch schreibt. Das Ergebnis wären möglicherweise halb geschriebene und damit unbrauchbare Informationen mit unabsehbaren Folgen.
   Dieses Problem löst die **Synchronisation** von Threads, um die es im Rest dieses Abschnitts 6.4 geht.
2. Über die Synchronisation hinaus muss sichergestellt sein, dass Thread 2 erst dann Daten aus A liest, wenn Thread 1 mit dem Schreiben fertig ist. Es geht also um die zeitliche Reihenfolge des Zugriffs.
   Den Schlüssel dazu liefert das Methodenpaar `wait` und `notify`, mit denen die Idee des „bedingten Wartens“ umgesetzt werden kann. Damit beschäftigt sich Abschnitt 6.6.

6.4.2 Gleichzeitiger Zugriff

Fehler beim unkontrollierten gleichzeitigen Zugriff

Das folgende Programm startet eine Anzahl Threads, von denen jeder eine gemein-

same, für alle Threads erreichbare Klassenvariable inkrementiert. Das Kommandozeilenargument legt die Anzahl der Threads fest, die sich am gemeinsamen Inkrementieren beteiligen.

```
public class ParallelIncrementer extends Thread {
    private static int count = 0;

    public void run() {
        for(int i = 0; i < 1_000_000; i++)
            count++;
        System.out.println(count);
    }

    public static void main(String... args) throws InterruptedException {
        for(int t = 0; t < Integer.parseInt(args[0]); t++)
            new ParallelIncrementer().start();
    }
}
```

Listing 6.23: Fehlerhaftes Hochzählen einer gemeinsamen Variablen in mehreren Threads.

Beim Start mit n Threads gibt das Programm n Zahlen aus. Die ersten Ausgaben lassen sich nicht vorhersagen, aber zuletzt sollte immer der Wert n Millionen erscheinen, weil zu diesem Zeitpunkt die Inkrementoperationen aller Threads ausgeführt sind. Das folgende Beispiel zeigt einen Programmaufruf mit drei Threads, der ein unerwartetes Ergebnis liefert:[39]

```
$ java ParallelIncrementer 3
1068707
1076899
2068707
```

Man hat den Eindruck, dass Java nicht mehr rechnen könnte. Die Erklärung ist aber eine andere: Die Threads inkrementieren `count` *gleichzeitig*, das heißt, sie führen gleichzeitig die Anweisung

```
count++;
```

aus. Der Compiler übersetzt diese Anweisung etwa wie die folgende:[40]

Einzelschritte einer scheinbar atomaren Anweisung

[39] Um diesen scheinbaren Fehler zu reproduzieren, sind möglicherweise mehrere Programmstarts und auf schnelleren Rechnern auch Starts mit höheren n nötig.

[40] Das stimmt so nicht ganz. Wäre es tatsächlich der Fall, dann könnte eine `short`- oder `byte`-Variable nicht inkrementiert werden, weil die Arithmetik auf der rechten Seite ein `int`-Ergebnis liefert, das

```
count = count + 1;
```

Diese Anweisung läuft in einzelnen Schritten ab:

1. den bisherigen Wert von `count` lesen,
2. den Wert um 1 erhöhen und
3. den inkrementierten Wert wieder in `count` speichern.

Mögliche Abfolgen der Einzelschritte

Ein Thread, der diese Anweisung ausführt, wickelt die drei Schritte genau in dieser Reihenfolge ab. Wenn aber *zwei* Threads A und B die Anweisung ausführen, können die sechs Schritte nacheinander ablaufen oder sich *beliebig vermischen!* Hier sind zwei mögliche Abfolgen abgedruckt. (Die Schreibweise X_n bedeutet, dass Thread X Schritt n ausführt.)

$A_1, A_2, A_3, B_1, B_2, B_3$
$A_1, B_1, A_2, B_2, A_3, B_3$

Die erste Abfolge der Schritte hinterlässt planmäßig den Wert 2 in der Variablen. Die zweite Abfolge führt aber zum Ergebnis 1, weil zuerst A_1 und B_1 den gleichen alten Wert 1 lesen, dann A_2 und B_2 beide auf 2 erhöhen und schließlich A_3 und B_3 *zweimal* nacheinander den gleichen Wert in `count` speichern. Es gibt viele verschiedene[41] Abfolgen der sechs Schritte, von denen einige zum richtigen, andere zum falschen Endergebnis führen.

Auftreten des Fehlers nicht vorhersagbar

Es ist vielleicht nicht sehr wahrscheinlich, dass zwei Inkrementanweisungen zeitlich so nah zusammentreffen, dass sich die Schritte in der oben gezeigten Art vermischen. Allerdings führt das Programm `ParallelIncrementer` sehr viele Inkrementanweisungen aus. Dabei kommt die problematische Schrittfolge einige Male vor und führt dann zu den beobachteten, scheinbar falschen Ausgaben.

▶ Der Compiler übersetzt Quelltext in Bytecode. Der Bytecode liegt in einer binären `class`-Datei vor, die für die effiziente Ausführung durch die JVM gebaut ist, aber nicht für menschliche Leser. Der **Java-Disassembler** `javap` (Teil des regulären JDK) gibt binären Bytecode einer `class`-Datei in Textdarstellung aus, wie im folgenden Beispiel:

```
$ javap -c ParallelIncrementer
```

nur mit einem expliziten Typecast wieder an die Variable auf der linken Seite zugewiesen werden kann. Der Compiler übersetzt die Inkrementanweisung so, dass kein Typecast nötig ist. An den drei Einzelschritten ändert das nichts.

41 Genau genommen gibt es 16 Abfolgen.

```
Compiled from "ParallelIncrementer.java"
public class ParallelIncrementer extends java.lang.Thread {
      ...
       9: getstatic     #2                  // Field count:I
      12: iconst_1
      13: iadd
      14: putstatic     #2                  // Field count:I
      ...
```

Hier ist nur der Ausschnitt der längeren Ausgabe abgedruckt, der der Anweisung `count++;` entspricht. Mit etwas scharfem Hinsehen kann man den Ablauf der Inkrementanweisung auf Bytecode-Ebene nachvollziehen:

1. Der Wert der Klassenvariablen `count` wird geholt. (9)
2. Die Konstante 1 wird bereitgelegt. (12)
3. Variablenwert und Konstante werden mit einer Integer-Addition verknüpft. (13)
4. Die Summe wird wieder in die Klassenvariable `count` geschrieben. (14)

Potenziell kann ein Thread nach jeder dieser Bytecode-Instruktionen unterbrochen werden. ◀

6.4.3 Zusammengesetzte Operationen

Allgemeines Problem zusammengesetzter Operationen

Die Ursache für das Problem von `ParallelIncrementer` (Listing 6.23) liegt im mehrteiligen Aufbau der Anweisung

```
count++;
```

Die Anweisung wirkt zwar im Quelltext wie eine Einheit, aber der Anschein täuscht.

Alle Einzelschritte zusammen bewirken die gewünschte Änderung. Wenn nur ein Teil der Einzelschritte ausgeführt ist, befindet sich das Programm in einem Zwischenzustand mit teilweise „alten“ und teilweise „neuen“ Daten. Andere Threads, die diesen Zwischenzustand beobachten, sehen ein inkonsistentes Bild.

Die Anweisung `count++;` ist ein verhältnismäßig einfacher Fall. Im Allgemeinen kann eine einzige logische Operation aus komplexen und aufwendigen Schritten bestehen, die erst bei vollständiger Abwicklung ein sinnvolles Ergebnis bewirken. Ohne Schutzmaßnahmen verschärft sich das Problem mit zunehmender Anzahl und Dauer der Schritte. Es wird dabei immer wahrscheinlicher, dass ein anderer Thread dazwischenkommt und auf einen inkonsistenten Zwischenzustand trifft.

Das Problem lässt sich lösen, wenn jeder einzelne Thread *ungestört alle Einzelschritte* einer zusammengesetzten Operation zu Ende bringen kann. Insbesondere dürfen andere Threads keine Gelegenheit erhalten, währenddessen die gemeinsamen Daten zu sehen oder gar zu verändern.

`synchronized`-Block

Kontrollstruktur mit Monitor-Objekt

Dieses Problem lösen `synchronized`-Blöcke, die syntaktisch einer Kontrollstruktur ähneln:[42]

```
synchronized(object) {
    statements
}
```

Das „Argument" im Kopf der `synchronized`-Anweisung ist ein beliebiges Objekt,[43] dessen Typ und Wert ohne Bedeutung sind. Jedes Java-Objekt verfügt über einen **Monitor**, den man sich als verborgene `boolean`-Objektvariable vorstellen kann. Um die Anweisungen *statements* auszuführen, muss ein Thread in den „Besitz" des Monitors gelangen. Ein Monitor ist nicht teilbar. Immer nur ein Thread kann ihn besitzen.

▶ Stellen Sie sich einen `synchronized`-Block wie einen versperrten Raum vor und den Monitor als Schlüssel zu diesem Raum, der an einem Schlüsselbrett neben der Tür hängt. Man kann den Raum nur durch diese eine Tür betreten und braucht dazu den Schlüssel.

- Kommt man an die Tür, während der Raum frei ist, dann nimmt man den Schlüssel, öffnet damit die Tür, betritt den Raum mit dem Schlüssel und sperrt hinter sich ab.
  Wenn man seine Arbeit im Raum erledigt hat, verlässt man den Raum und hängt den Schlüssel wieder zurück an das Schlüsselbrett.
- Kommt man an die Tür, während der Raum belegt und folglich versperrt ist, dann fehlt der Schlüssel. Man muss warten, bis der andere den Raum wieder verlässt und den Schlüssel an das Schlüsselbrett zurückbringt.

◀

Exklusive Ausführung des Blocks

Sobald ein Thread einen `synchronized`-Block erreicht, fordert er den Monitor an.

[42] Das Schlüsselwort `synchronized` muss von geschweiften Klammern gefolgt werden. Das ist eine kleine Anomalie der Syntax. Bei Kontrollstrukturen können die geschweiften Klammern weg bleiben, wenn nur eine einzige Anweisung untergeordnet ist. `synchronized`, `try/catch/finally` und einige weitere Konstrukte erfordern dagegen *immer* geschweifte Klammern.

[43] `null` ist eine Referenz, aber *kein* Objekt und deshalb nicht zulässig.

- Ist der Monitor **frei**, dann nimmt ihn der Thread „in Besitz“ und darf den Rumpf ausführen. Beim Verlassen des Rumpfes stellt der Thread den Monitor automatisch wieder zur Verfügung.
- Ist der Monitor gerade an einen anderen Thread **vergeben**, dann blockiert der Thread so lange, bis der Monitor wieder zurückgegeben wird und in Besitz genommen werden kann.
- **Besitzt** der Thread den Monitor bereits, kann er den Rumpf ausführen.

Mit diesem Verfahren ist sichergestellt, dass zu einem Zeitpunkt *immer höchstens einer* von mehreren Threads, die den betreffenden Monitor anfordern, den Rumpf des `synchronized`-Blocks ausführt. Dieser eine Thread ist der alleinige, exklusive „Besitzer“ des Monitors. Der Monitor lässt sich nicht kopieren, um damit beispielsweise unzulässigerweise einem weiteren Thread Zugang zu einem „besetzten“ `synchronized`-Block zu verschaffen.

Kein Ausschluss unbeteiligter Threads

Andere Threads, die überhaupt keine oder andere Monitore verlangen, schließt dieses Verfahren nicht aus. Sie können den `synchronized`-Block zu beliebigen Zeitpunkten ungehindert ausführen. Hierin liegt eine potenzielle Fehlerquelle: Der Entwickler muss sicherstellen, dass konkurrierende Threads tatsächlich am *selben* Objekt synchronisieren! Weder Compiler noch JVM erkennen fehlerhafte Synchronisation.

JVM regelt Freigabe des Monitors

Das Zuweisen und Freigeben von Monitoren erledigt die JVM. Benutzercode hat darauf keinen direkten Einfluss. Das garantiert, dass beim Verlassen eines `synchronized`-Blocks der beim Eintritt übernommene Monitor zuverlässig zurückgegeben wird.[44] Es kann nicht vorkommen, dass ein Monitor irrtümlich weiter behalten wird.[45]

Das folgende Programm behebt die Probleme von `ParallelIncrementer`. Die Anweisung `count++;` ist jetzt durch `synchronized` geschützt. Zur Synchronisierung dient die Klassenvariable `monitor`, die für alle Threads sichtbar ist.

```
public class ParallelSyncedIncrementer extends Thread {
    private static Integer count = 0;

    private static final Object monitor = new Object();

    public void run() {
        for(int i = 0; i < 1_000_000; i++)
```

[44] Das gilt nicht, wenn der Monitor beim Eintritt in den `synchronized`-Block schon dem Thread gehörte. In diesem Fall bleibt er im Besitz des Threads, bis schließlich der äußere `synchronized`-Block mit diesem Monitor verlassen wird.

[45] Das Package `java.util.concurrent` enthält viele nützliche Hilfsmittel zur Lösung fortgeschrittener Synchronisationsaufgaben. Die Klasse `Lock` in diesem Package repräsentiert eine Art Monitor, die explizit vom Programmierer belegt und freigegeben werden kann.

```
            synchronized(monitor) {
                count++;
            }
        System.out.println(count);
    }

    public static void main(String... args) throws InterruptedException {
        for(int t = 0; t < Integer.parseInt(args[0]); t++)
            new ParallelSyncedIncrementer().start();
    }
}
```

Listing 6.24: Fehlerfreies Hochzählen einer gemeinsamen Variablen in mehreren Threads.

Das Programm zählt jetzt zuverlässig hoch, unabhängig von der Anzahl der gestarteten Threads.[46]

Monitor-Objekte

Jedes Objekt geeignet als Monitor

In Java verfügt *jedes* Objekt automatisch über einen Monitor, um den sich Threads bewerben können. Typ und Wert spielen keine Rolle. Selbst ein Exemplar der Klasse `Object`, wie im vorhergehenden Beispiel, reicht dazu aus.

Ausschluss nur mit dem*selben* Objekt

Trotzdem ist Vorsicht geboten: Nur wenn Threads um den Monitor des*selben* (nicht: des *gleichen*) Objekts konkurrieren, können sie mit `synchronized` kontrolliert werden. Das Programm `ParallelSyncedIncrementer` (Listing 6.24) funktioniert zum Beispiel nicht mehr, wenn `monitor` als Objekt- und nicht als Klassenvariable definiert ist:

```
private Object monitor = new Object();     // ohne "static"
```

Jetzt verfügt jedes `Thread`-Objekt über seine eigene Objektvariable `monitor`. Beim Eintritt in den `synchronized`-Block bewirbt sich jeder Thread um den Monitor *seiner eigenen Objektvariablen* und erhält ihn natürlich sofort, weil sich kein anderer Thread dafür interessiert. Der `synchronized`-Block ist in diesem Fall wirkungslos und es kommt zu den gleichen Fehlern wie im unsynchronisierten Programm `ParallelIncrementer` (Listing 6.23).

Globale Objekte nur bedingt geeignet

Im Beispiel `ParallelSyncedIncrementer` (Listing 6.24) reicht eine Klassenvariable als Monitor aus. In realen Programmen behindern Klassenvariablen aber oft die Fortentwicklung. Deshalb übergibt man konkurrierenden Thread-Objekten besser

[46] Das Programm gibt zwischendurch auch andere Zahlen als Vielfache von einer Million aus. Das ist in Ordnung. Entscheidend ist die zuletzt ausgegebene Zahl, die n Millionen lauten muss.

ein und dasselbe Objekt als Konstruktorargument, das die Threads als Monitor verwenden.

Nicht alle Typen sind gleichermaßen geeignet für Monitor-Objekte. Wrapper-Objekte (Objekte der Typen `Integer`, `Character`, `Long` und so weiter) sind zum Beispiel schlechte Kandidaten. Sie sind für „kleine Werte“ eindeutig, für größere nicht. Die Identität dieser Objekte ist dabei schwer nachzuvollziehen.

Vorsicht mit Wrapper-Objekten

Daran scheitert auch die eigentlich naheliegende Idee, in `ParallelSyncedIncrementer` (Listing 6.24) den Zähler selbst als Wrapper-Objekt zu definieren und dieses als Monitor-Objekt zu verwenden:

```
private static Integer count = 0;  // statt "int"
...
synchronized(count) {...           // statt "monitor"
```

Ein Inkrement von `count` ändert nicht das vorhandene `Integer`-Objekt, sondern weist ein neues an `count` zu.

Schwer auffindbare Fehler durch falsche Monitor-Objekte

Fehler dieser Art sind schwer zu finden. Der Compiler sieht kein Problem und übersetzt den Quelltext klaglos. Das Programm läuft auch manchmal wie erwartet ab, scheitert aber in anderen Fällen ohne erkennbare Ursache. Darüber hinaus lässt sich der Fehler nicht zuverlässig reproduzieren.

6.4.4 Synchronisation auf einem Monoprozessor

Time-Slicing auf Monoprozessoren

Auf einem einzelnen Prozessor gelten etwas andere Voraussetzungen: Dort können keine zwei Threads *gleichzeitig* die Inkrementanweisung ausführen, weil immer nur ein Thread tatsächlich rechnet. Leider ist die Annahme falsch, dass die oben beschriebenen Probleme mit der parallelen Ausführung desselben Codes hier nicht auftreten könnten. Ein Programm wie `ParallelIncrementer` (Listing 6.23) funktioniert auch auf einem einzelnen Prozessor nicht zuverlässig.

Um die Ursachen zu verstehen, muss man sich das Time-Slicing (siehe Seite 394) des Schedulers genauer ansehen: Er lässt Threads abwechselnd für je eine bestimmte Zeit rechnen, um eine gewisse Fairness zu gewährleisten. Ein Thread kann allerdings weder steuern noch vorhersehen, *wann genau* ihm der Scheduler den Prozessor entzieht und ein anderer Thread an die Reihe kommt. Der Scheduler arbeitet dabei auf der Ebene von Bytecode-Instruktionen, nicht auf der Ebene von Anweisungen im Quelltext.[47] Obwohl hier nur ein einziger Prozessor arbeitet, ergeben

[47] Der Quelltext wurde ja übersetzt. Anweisungsstrukturen lösen sich dabei weitgehend auf.

sich daraus die gleichen Probleme, wie der gleichzeitige Zugriff von mehreren Prozessoren (Seite 404).

Synchronisation zusammengesetzter Operationen ist also *immer* nötig, wenn sie von mehr als einem Thread ausgeführt werden. Dabei ist es unerheblich, wie viele Prozessoren im Spiel sind.

6.4.5 Atomare Operationen

Atomare Zuweisungen einiger primitiver Typen

`synchronized` garantiert, dass die Anweisungen im Block *ohne Unterbrechung* durch andere Interessenten bis zum Ende durchlaufen. Solche Operationen bezeichnet man als **atomar** im Sinne von (zeitlich) „unteilbar". Anweisungen ohne `synchronized` können nach teilweiser Ausführung an beliebigen Punkten unterbrochen oder die dabei erreichten Zwischenzustände nach außen sichtbar werden.

Einige sehr einfache Operationen führt die JVM auch ohne Synchronisation immer atomar aus. Dazu zählen Zuweisung[48] und Vergleich primitiver Typen, außer `long` und `double`,[49] und von Referenzvariablen.

Auf den äußeren Anschein im Quelltext ist dabei kein Verlass. Am `ParallelIncrementer` (Listing 6.23) ist beispielsweise zu sehen, dass die täuschend kompakte Inkrementanweisung

```
count++;
```

nicht atomar abläuft.[50]

6.4.6 Länge von `synchronized`-Blöcken

`synchronized`-Blöcke erzwingen sequenzielle Verarbeitung

`synchronized` sichert zeitweise exklusiven „Besitz" eines Codeabschnitts zu. Das

[48] Wohl gemerkt: Nur das eigentliche *Schreiben* eines neuen Werts in eine Variable verläuft atomar, aber nicht die vorhergehende Berechnung des Werts gemäß der rechten Seite der Wertzuweisung.

[49] Das bedeutet, dass eine Zuweisung wie `long n = 1;` unterbrochen werden kann. Ein `long`-Wert umfasst 64 Bits. Der Scheduler kann den Thread nach Eintrag der ersten 32 Bits unterbrechen. Der Wert der Variablen besteht dann zur Hälfte noch aus „alten" Bits, zur Hälfte schon aus „neuen" Bits und ist voraussichtlich völlig sinnlos. Der Modifier `volatile` (Seite 422) ändert dieses Verhalten.

[50] Die Klasse `AtomicInteger` im Package `java.util.concurrent` kapselt `int`-Werte und definiert Operationen mit dem gekapselten Wert, die in synchronisierten Methoden implementiert sind und deshalb „atomar" ablaufen.

bedeutet, dass zu einem Zeitpunkt *höchstens ein* Thread den Rumpf eines `synchronized`-Blocks ausführt. Mehrere Threads durchlaufen den Block also nicht parallel, sondern *sequenziell*. Dem Ziel der Nutzung mehrerer Prozessoren wirkt das allerdings entgegen.

Das folgende Programm `ParallelFibonacciAdder` startet eine Anzahl Threads, von denen jeder die ersten vierzig Fibonaccizahlen (siehe Seite 256) zur gemeinsamen Klassenvariablen `count` addiert.[51]

```
public class ParallelFibonacciAdder extends Thread {
    private static int count = 0;

    private static final Object monitor = new Object();

    private static final Fibonacci fibonacci = new Fibonacci();

    public void run() {
        for(int i = 0; i < 40; i++)
            synchronized(monitor) {
                count += fibonacci.fib(i);
            }
        out.println(count);
    }

    public static void main(String... args) throws InterruptedException {
        for(int t = 0; t < Integer.parseInt(args[0]); t++)
            new ParallelFibonacciAdder().start();
    }
}
```

Listing 6.25: Parallele Berechnung von Fibonaccizahlen in mehreren Threads.

Die kritische Operation ist synchronisiert, daher arbeitet das Programm fehlerfrei:[52]

```
$ java ParallelFibonacciAdder 1
165580141
$ java ParallelFibonacciAdder 4
174807606
331160282
535828592
662320564
```

[51] Man könte auch das `Fibonacci`-Objekt als Monitor-Objekt verwenden. Es ist eindeutig und für alle Threads dasselbe. Das Beispiel verwendet ein eigenes Monitor-Objekt, um die völlig unterschiedlichen Rollen zu unterstreichen.

[52] Es kommt nur auf die letzte Zahl an. Diese ist korrekt, denn $662320564 = 4 \cdot 165580141$. Dieses Programm ändert die gemeinsame Variable `count` viel seltener als die vorhergehenden Beispielprogramme, die sie praktisch ständig inkrementieren. Daher ist hier auch ohne Schutz die Wahrscheinlichkeit gleichzeitiger Zugriffe nicht sehr groß, aber immer noch vorhanden. Ohne Synchronisation wäre der Programmablauf daher ein Glücksspiel, wenn auch ein Erfolg versprechendes.

Übersynchronisierung

Zu großer `synchronized`-Block ruiniert Parallelisierung

Der `ParallelFibonacciAdder` (Listing 6.25) liefert die erwarteten Ergebnisse. Das folgende Programm ergänzt die `run`-Methode aber um die Ausgabe der Laufzeit:

```
public void run() {
    long start = currentTimeMillis();
    for(int i = 0; i < 40; i++)
        synchronized(monitor) {
            count += fibonacci.fib(i);
        }
    long millis = currentTimeMillis() - start;
    out.printf("%d, %d millis%n", count, millis);
}
```

Listing 6.26: Ergänzung um Ausgabe der Laufzeit jedes Threads.

Hier stellt sich heraus, dass das Programm von mehreren Prozessoren in keiner Weise profitiert. Die folgenden Ergebnisse erhält man auf einer Quadcore-CPU:

```
$ java ParallelFibonacciAdder 1
165580141, 730 millis
$ java ParallelFibonacciAdder 4
165580149, 710 millis
331160291, 1532 millis
496740431, 2237 millis
662320564, 2944 millis
```

Vier Threads brauchen rund viermal so lange wie ein einzelner Thread. Von Parallelisierung keine Spur!

`ParallelFibonacciAdder` (Listing 6.25) ist **übersynchronisiert**. Der `synchronized`-Block schließt nicht nur die kritische Addition zum gemeinsamen Zähler ein, sondern auch die langwierige Berechnung der Fibonaccizahl. Diese Berechnung arbeitet aber nur mit lokalen Daten innerhalb des Threads und konkurriert überhaupt nicht mit anderen Threads. Sie muss nicht unter gegenseitigem Ausschluss ablaufen.

Minimale Synchronisation

Ziel: `synchronized`-Blöcke minimaler Länge

Ein `synchronized`-Block sollte *nicht länger als nötig* sein. Er muss alle Teile einer kritischen Operation umfassen, aber auch nicht mehr. Unnötig lange `synchronized`-Blöcke beeinträchtigen zwar nicht die Korrektheit eines Programms, machen aber die Vorteile der Nebenläufigkeit zunichte.

Übertragen auf `ParallelFibonacciAdder` (Listing 6.25) bedeutet das, dass die Fibonaccizahlen *außerhalb* des `synchronized`-Blocks berechnet werden sollten. Nur die eigentliche Addition zum gemeinsamen Zähler ist zu schützen:

```
public class ParallelFibonacciFastAdder extends Thread {
    private static int count = 0;

    private static final Object monitor = new Object();

    private static final Fibonacci fibonacci = new Fibonacci();

    public void run() {
        long start = System.currentTimeMillis();
        for(int i = 0; i < 40; i++) {
            long f = fibonacci.fib(i);
            synchronized(monitor) {
                count += f;
            }
        }
        long millis = System.currentTimeMillis() - start;
        System.out.printf("%d, %d millis%n", count, millis);
    }

    public static void main(String... args) throws InterruptedException {
        for(int t = 0; t < Integer.parseInt(args[0]); t++)
            new ParallelFibonacciFastAdder().start();
    }
}
```

Listing 6.27: Optimal synchronisierte parallele Berechnung von Fibonaccizahlen in mehreren Threads.

Ein Start von `ParallelFibonacciFastAdder` auf einer Quadcore-CPU zeigt, dass alle Prozessoren ausgelastet sind, weil die sequenzielle Addition im Verhältnis zur parallelen Berechnung der Fibonaccizahlen nicht ins Gewicht fällt. Das Programm bremst daher nicht nennenswert ab, bis die Anzahl der Threads die Anzahl der Prozessoren erreicht:

```
$ java ParallelFibonacciFastAdder 1
165580141, 774 millis
$ java ParallelFibonacciFastAdder 4
472582606, 775 millis
535828592, 782 millis
599074578, 788 millis
662320564, 791 millis
```

6.4.7 `synchronized`-Methoden

In vielen Fällen bietet es sich an, einen exklusiv auszuführenden Codeabschnitt in

Kurzschreibweise für Methode mit synchronisiertem Rumpf

eine eigene Methode auszulagern. Als Monitor eignet sich dabei oft das Zielobjekt des Methodenaufrufs selbst, das die Pseudovariable `this` referenziert.

Weil diese Anordnung von `synchronized`-Blöcken häufig gebraucht wird, bietet Java eine vereinfachte Schreibweise dafür an. Eine Methode mit dem Aufbau

```
type method(parameters) {
   synchronized(this) {
      ...
   }
}
```

kann kürzer mit `synchronized` als Modifier geschrieben werden als

```
 synchronized type method(parameters) {
   ...
}
```

Voraussetzungen für `synchronized`-Methoden

Der Unterschied ist rein syntaktisch und wird vom Compiler eingeebnet. Die beiden Voraussetzungen für den Einsatz des Modifiers `synchronized` sind:

1. Der gesamte Methodenrumpf soll geschützt werden und
2. Monitor-Objekt ist `this`, das heißt das Zielobjekt des Aufrufs.

Der Modifier `synchronized` spielt weder für die Redefinition noch für das Überladen von Methoden eine Rolle. In Interfaces ist `synchronized` nicht sinnvoll und nicht erlaubt.

Anwendungsbeispiele für `synchronized`-Methoden folgen weiter unten (Seite 422).

Statisch synchronisierte Methoden

Typobjekt als Monitor

Auch statische Methoden können mit `synchronized` vor paralleler Ausführung geschützt werden. Als Monitor-Objekt dient in diesem Fall das Typobjekt der Klasse. Die Definition

```
class SomeClass {
   static synchronized type method(parameters) {
      ...
   }
}
```

wird übersetzt wie

```
class SomeClass {
   static type method(parameters) {
       synchronized(SomeClass.class) {
           ...
       }
   }
}
```

Typobjekte sind Unikate, das heißt, es gibt zu jeder Klasse genau ein einziges Exemplar. Statisch synchronisierte Methoden sind folglich global synchronisiert. Aufrufe laufen in der gesamten JVM exklusiv ab.

Das Programm `ParallelFibonacciFastAdder` (Listing 6.27) vereinfacht sich mit einer statisch synchronisierten Methode `add` zum Inkrementieren des Zählers folgendermaßen:

Globale Synchronisation

```
public class ParallelSyncmethodAdder extends Thread {
    private static int count = 0;

    private static final Fibonacci fibonacci = new Fibonacci();

    private static synchronized void add(long f) {
        count += f;
    }

    public void run() {
        long start = System.currentTimeMillis();
        for(int i = 0; i < 40; i++)
            add(fibonacci.fib(i));
        long millis = System.currentTimeMillis() - start;
        System.out.printf("%d, %d millis%n", count, millis);
    }

    public static void main(String... args) throws InterruptedException {
        for(int t = 0; t < Integer.parseInt(args[0]); t++)
            new ParallelSyncmethodAdder().start();
    }
}
```

Listing 6.28: Einsatz einer `synchronized`-Methode.

Ohne den Modifier `static` für `add` arbeitet das Programm nicht zuverlässig, weil die Methode dann `this`, das Zielobjekt des Aufrufs, als Monitor verwendet. Dieses Zielobjekt ist aber in jedem Thread ein anderes, nämlich der jeweilige Thread selbst. Der gegenseitige Ausschluss beim Aufruf von `add` durch die verschiedenen Threads ist nicht mehr gewährleistet!

6.5 `volatile` und Deadlocks

6.5.1 Anwendungsbeispiel Parkhaus

Durchgehendes Beispiel für den Rest des Kapitels

Als Anwendungsbeispiel für diesen Abschnitt dient eine einfache Simulation eines Parkhauses. Auch die verbleibenden Themen dieses Kapitels lassen sich gut anhand des Parkhaus-Beispiels veranschaulichen, deshalb wird es an dieser Stelle etwas ausführlicher vorgestellt.

Die Anwendung soll ein Parkhaus simulieren, in das Autos einfahren, darin eine Weile parken und es schließlich wieder verlassen. Das folgende Interface definiert die beiden Methoden eines Parkhauses:[53]

```
public interface Parkhaus {
    void enter(Auto auto) throws InterruptedException;

    void leave(Auto auto);
}
```

Listing 6.29: Interface für verschiedene Implementierungen von Parkhäusern.

Autos mit festem Verhalten als Threads

Das Parkhaus ist passiv und keine `Thread`-Klasse. Aktiv sind die Autos, die das Parkhaus benutzen. Sie werden mit Objekten der Klasse `Auto` modelliert, die von `Thread` abgeleitet ist. Alle Autos zeigen grundsätzlich das gleiche Verhalten, nur dauern die einzelnen Schritte unterschiedlich lang. Das Verhalten ist in der `run`-Methode implementiert:

1. Nach dem Start ist ein Auto eine gewisse Zeit unterwegs, bis es am Parkhaus ankommt. Die Objektvariable `anfahrt` legt diese Zeit fest.
2. Sobald es am Parkhaus ankommt, ruft es die Methode `enter` auf, um in das Parkhaus zu gelangen.
3. Im Parkhaus parkt es für eine bestimmte Dauer, die die Objektvariable `parkzeit` festlegt.
4. Nach Ablauf der Parkzeit ruft es die Methode `leave` auf, um das Parkhaus wieder zu verlassen. Dann spielt das Auto für die Simulation keine Rolle mehr. Die `run`-Methode endet.

Der `Auto`-Konstruktor erwartet als Argumente das Parkhaus, das das Auto benutzen soll, und die beiden Zeiten `anfahrt` und `parkzeit`, jeweils in Millisekunden.

[53] Die Exceptionsignatur der Methode `enter` wird weiter unten gebraucht und spielt hier noch keine Rolle.

```
public class Auto extends Thread {

    private final Parkhaus parkhaus;

    private final int anfahrt;

    private final int parkzeit;

    public Auto(Parkhaus parkhaus, int anfahrt, int parkzeit) {
        this.parkhaus = parkhaus;
        this.anfahrt = anfahrt;
        this.parkzeit = parkzeit;
    }

    @Override public void run() {
        try {
            Thread.sleep(anfahrt);
            parkhaus.enter(this);
            Thread.sleep(parkzeit);
            parkhaus.leave(this);
        }
        catch(InterruptedException ie) {
            throw new AssertionError(ie);
        }
    }

}
```

Listing 6.30: Autos, die ein Parkhaus anfahren und dort eine Weile parken.

Interrupts sind in dieser Simulation nicht vorgesehen. Eine `InterruptedException` kann daher eigentlich nicht auftreten. Falls doch, stoppt das Programm mit einem `AssertionError`.

Protokoll von Methodenaufrufen mit Zeitmarken

In dieser Anwendung spielt der genaue zeitliche Ablauf eine große Rolle. Er lässt sich mit zusätzlichen Protokollausgaben in der `run`-Methoden besser nachvollziehen. Diese Ausgaben ändern nichts an der Arbeitsweise der Methode, sondern dienen nur zur nachträglichen Überprüfung:

```
@Override public void run() {
    try {
        log(this + " fährt los");
        Thread.sleep(anfahrt);
        log(this + " kommt an, will einfahren");
        parkhaus.enter(this);
        log(this + " ist drin, parkt");
        Thread.sleep(parkzeit);
        log(this + " ist fertig, will ausfahren");
        parkhaus.leave(this);
        log(this + " ist draußen, fährt weg");
    }
    catch(InterruptedException ie) {
        throw new AssertionError(ie);
```

```
    }
}
```

Listing 6.31: `run`-Methode der Klasse `Auto` mit zusätzlichen Protokollausgaben.

Die statische Methode `log` ist an anderer Stelle definiert. Sie gibt das String-Argument zusammen mit einer Zeitmarke auf dem Bildschirm aus. Die Definition von `log` spielt hier keine Rolle.

Eine erste Implementierung eines Parkhauses protokolliert lediglich die Ein- und Ausfahrt von Autos:

```
public class SimpleParkhaus implements Parkhaus {
    @Override public void enter(Auto auto) throws InterruptedException {
        log(auto + " fährt ein");
    }

    @Override public void leave(Auto auto) {
        log(auto + " fährt aus");
    }

}
```

Listing 6.32: Minimale Implementierung eines unbegrenzten Parkhauses.

Ein Hauptprogramm erzeugt zuerst ein `Parkhaus`-Objekt und schickt dann ein paar Autos auf die Reise. Anfahrt- und Parkzeit jedes Autos holt das Hauptprogramm von der Kommandozeile:

```
public static void main(String[] args) {
    Parkhaus parkhaus = new SimpleParkhaus();
    for(String arg: args) {
        String[] words = arg.split("\\D+");
        new Auto(parkhaus,
                 Integer.parseInt(words[0]),
                 Integer.parseInt(words[1])).start();
    }
}
```

Listing 6.33: Hauptprogramm mit einem Auto pro Kommandozeilenargument *fahrzeit*/*parkzeit*.

Der Programmstart zeigt den erwarteten Ablauf:

```
$ java SimpleParkhaus 1000/1000 1500/1000 2000/1000
[  0.0] Auto#1 fährt los
[  0.0] Auto#3 fährt los
```

```
[  0.0] Auto#2 fährt los
[  1.0] Auto#1 kommt an, will einfahren
[  1.0] Auto#1 fährt ein
[  1.0] Auto#1 ist drin, parkt
[  1.5] Auto#2 kommt an, will einfahren
[  1.5] Auto#2 fährt ein
[  1.5] Auto#2 ist drin, parkt
[  2.0] Auto#3 kommt an, will einfahren
[  2.0] Auto#3 fährt ein
[  2.0] Auto#3 ist drin, parkt
[  2.0] Auto#1 ist fertig, will ausfahren
[  2.0] Auto#1 fährt aus
[  2.0] Auto#1 ist draußen, fährt weg
[  2.5] Auto#2 ist fertig, will ausfahren
[  2.5] Auto#2 fährt aus
[  2.5] Auto#2 ist draußen, fährt weg
[  3.0] Auto#3 ist fertig, will ausfahren
[  3.0] Auto#3 fährt aus
[  3.0] Auto#3 ist draußen, fährt weg
```

6.5.2 Optimierung von Variablenzugriffen

Parkhaus mit begrenztem Fassungsvermögen

Eine etwas sinnvollere Fassung des Beispiels begrenzt die Größe des Parkhauses. An einem vollen Parkhaus ankommende Autos müssen warten, bis ein anderes Auto ausfährt und einen Platz freimacht. Die folgende Klasse erweitert `SimpleParkhaus` (Listing 6.32) um eine Objektvariable `autos`, die die Anzahl der augenblicklich parkenden Autos mitzählt. Die Methoden `enter` und `leave` erhöhen beziehungsweise verringern diesen Zähler.

Warteschleife zur Verzögerung von Aufrufen

Die `final`-Objektvariable `capacity` speichert das unveränderliche Fassungsvermögen des Parkhauses. `enter` vergleicht die aktuelle Anzahl Autos im Parkhaus mit diesem Fassungsvermögen und wartet gegebenenfalls in einer Schleife, solange das Parkhaus voll belegt ist. Der Aufrufer von `enter` wird blockiert, bis ein Platz frei geworden ist. Im folgenden Beispiel verfügt das Parkhaus nur über einen einzigen Parkplatz. Das ist zwar etwas unrealistisch, vereinfacht aber die Untersuchung der Abläufe und fördert potenzielle Probleme schneller zutage.

```
public class LimitedParkhaus implements Parkhaus {
    private final int capacity = 1;

    private int autos = 0;

    public void enter(Auto auto) throws InterruptedException {
        // Aufrufer blockieren, bis wieder Platz ist
        while(autos == capacity)
            ;
        log(auto + " fährt ein");
        autos++;
    }
```

```
    public void leave(Auto auto) {
        log(auto + " fährt aus");
        autos--;
    }

    public static void main(String[] args) {
        System.out.println(Runtime.getRuntime().availableProcessors() + " processor(s)");
        Parkhaus parkhaus = new LimitedParkhaus();
        for(String arg: args)
            // ...
    }
}
```

Listing 6.34: Parkhaus mit einer begrenzten Anzahl vom Plätzen.

Korrekter Ablauf auf einem Monoprozessor

Auf einem System mit einem einzigen Prozessor läuft das Programm ab wie geplant:

```
$ java LimitedParkhaus 1000/1000 1500/1000 2000/1000
1 processor(s)
[  0.0] Auto#3 fährt los
[  0.0] Auto#1 fährt los
[  0.0] Auto#2 fährt los
[  1.0] Auto#1 kommt an, will einfahren
[  1.0] Auto#1 fährt ein
[  1.0] Auto#1 ist drin, parkt
[  1.5] Auto#2 kommt an, will einfahren
[  2.0] Auto#1 ist fertig, will ausfahren
[  2.0] Auto#1 fährt aus
[  2.0] Auto#1 ist draußen, fährt weg
[  2.0] Auto#2 fährt ein
[  2.0] Auto#2 ist drin, parkt
[  2.0] Auto#3 kommt an, will einfahren
[  3.1] Auto#2 ist fertig, will ausfahren
[  3.1] Auto#2 fährt aus
[  3.1] Auto#2 ist draußen, fährt weg
[  3.1] Auto#3 fährt ein
[  3.1] Auto#3 ist drin, parkt
[  4.1] Auto#3 ist fertig, will ausfahren
[  4.1] Auto#3 fährt aus
[  4.1] Auto#3 ist draußen, fährt weg
```

Blockieren auf einem Multiprozessor

Auf einem System mit mehreren Prozessoren startet das Programm zwar, reagiert aber plötzlich nicht mehr:

```
$ java LimitedParkhaus 1000/1000 1500/1000 2000/1000
8 processor(s)
[  0.0] Auto#1 fährt los
[  0.0] Auto#3 fährt los
[  0.0] Auto#2 fährt los
```

```
[  1.0] Auto#1 kommt an, will einfahren
[  1.0] Auto#1 fährt ein
[  1.0] Auto#1 ist drin, parkt
[  1.5] Auto#2 kommt an, will einfahren
[  2.0] Auto#3 kommt an, will einfahren
[  2.0] Auto#1 ist fertig, will ausfahren
[  2.0] Auto#1 fährt aus
[  2.0] Auto#1 ist draußen, fährt weg
(keine Reaktion mehr)
```

Cachen von Variablenwerten in jedem Thread

Der Grund für dieses unerwartete Verhalten ist eine an sich sehr wirksame Optimierung im Java-Code: Wenn ein Variablenwert mehrmals nacheinander verwendet wird, merkt sich Java den Wert beim ersten Lesen und arbeitet dann mit dem gemerkten Wert weiter, *ohne erneut auf die Variable zuzugreifen*.[54] Eine Wertzuweisung löscht den gemerkten Wert natürlich.

Den Bytecode mit dieser Optimierung erzeugt der Java-Compiler nach einer gründlichen Analyse des Quelltexts. Der Compiler kann aber das spätere Laufzeitverhalten nicht vorhersehen und daher auch keine möglicherweise parallel laufenden Threads vorausahnen.[55] In der Klasse `LimitedParkhaus` (Listing 6.34) betrifft das die Warteschleife in der Methode `enter`:

```
while(autos == capacity)
   ;
```

Die Methode holt den Wert der Objektvariablen `autos` nur einmal und fährt dann mit diesem Wert fort, ohne die Variable später noch einmal zu lesen! Wenn die Bedingung also beim Erreichen der Schleife zutrifft, ändert sich daran nichts mehr. Dass `autos` in der Zwischenzeit von einem anderen Thread geändert werden könnte, entgeht der Warteschleife. Sie kommt zu keinem Ende und der Aufruf der `enter`-Methode kehrt nie zurück.

Löschen des Caches beim Time-Slicing

Ein Thread merkt sich Variablenwerte nur, solange er im Zustand *Running* (siehe Seite 387) bleibt. Aus diesem Grund funktioniert das Programm auf einem System mit einem einzigen Prozessor: Das Time-Slicing verdrängt die Threads regelmäßig aus dem Zustand *Running* und löst damit mittelbar eine Aktualisierung der Variablenwerte aus.

Abgesehen davon löscht ein Thread alle gemerkten Variablenwerte beim Erreichen und Verlassen von `synchronized`-Blöcken und -Methoden.

[54] Diese Optimierung leistet einen ganz wesentlichen Beitrag zur Performance von Java und kann keineswegs als verzichtbarer Luxus abgetan werden. Häufig benutzte Werte verbleiben in Registern der JVM, die besonders schnell erreichbar sind. Gewöhnliche Variablen liegen dagegen im Hauptspeicher, auf den nur vergleichsweise langsam zugegriffen werden kann.

[55] Um das Problem kategorisch auszuschließen, müsste der Compiler bei praktisch jedem Codefragment mit paralleler Ausführung rechnen und folglich die Optimierung überhaupt aufgeben.

6.5.3 Modifier `volatile`

Explizite Ausnahme von der Optimierung

Der Modifier `volatile` schaltet die Optimierung des Compilers für die betreffende Variable gezielt ab. Der Wert einer so markierten Variablen wird beim Lesen *jedes Mal aufs Neue* aus der Variablen geholt. Das kostet zwar ein wenig Performance, beseitigt aber die oben aufgedeckten Probleme. Mit `volatile` gibt der Entwickler dem Compiler eine entscheidende Information über den Quelltext, die der Compiler alleine nicht findet.

Ergänzt man in `LimitedParkhaus` (Listing 6.34) die Definitionen von `autos` um den Modifier `volatile`,

```
private  volatile int autos = 0;
```

dann darf kein Thread diesen Variablenwert mehr zwischenspeichern. Jeder Thread muss den Wert bei jedem Zugriff direkt aus der Variablen `autos` holen. Die Warteschleife

```
while(autos == capacity)
   ;
```

endet planmäßig, wenn ein anderer Thread den Wert von `autos` verringert. Das Beispielprogramm arbeitet damit korrekt, unabhängig von der Anzahl der Prozessoren.

Atomare Wertzuweisung

Atomare Wertzuweisung bei allen Typen

`volatile` hat noch eine weitere Auswirkung: Wertzuweisungen so definierter Variablen sind *immer* atomar. Das ist insbesondere für `long`- und `double`-Variablen interessant, deren Zuweisung andernfalls unterbrochen werden könnte (siehe Seite 410).

Der Modifier `volatile` ist nur bei veränderlichen Objektvariablen sinnvoll und auch nur dort zulässig. Lokale Variablen und Parameter werden ohnedies nicht zwischen Threads geteilt, sodass das ganze Problem nicht auftritt. Bei unveränderlichen Objektvariablen ist die Frage paralleler Modifikationen ebenfalls gegenstandslos.

6.5.4 Deadlocks

Mangelnde Synchronisation trotz `volatile`

Die Klasse `LimitedParkhaus` (Listing 6.34) arbeitet selbst mit der Ergänzung um

den Modifier `volatile` nicht zuverlässig. Elementare Zusicherungen in den Methoden `enter` und `leave` decken den Fehler auf:

```
public class CheckedParkhaus implements Parkhaus {
    private final int capacity = 1;

    private volatile int autos = 0;

    public void enter(Auto auto) throws InterruptedException {
        while(autos == capacity)
            ;
        log(auto + " fährt ein");
        autos++;
        assert autos > 0: "Parkhaus nicht leer";
    }

    public void leave(Auto auto) {
        log(auto + " fährt aus");
        autos--;
        assert autos < capacity: "Parkhaus hat Platz";
    }

    public static void main(String[] args) {
        // ...
    }
}
```

Listing 6.35: Parkhaus mit Zusicherungen.

Startet man dieses Programm mit einigen Autos, die *gleichzeitig* am Parkhaus ankommen, dann scheitern die Zusicherungen:

```
$ java -ea CheckedParkhaus 1000/1000 1000/1000 1000/1000
8 processor(s)
[  0.0] Auto#2 fährt los
[  0.0] Auto#1 fährt los
[  0.0] Auto#3 fährt los
[  1.0] Auto#2 kommt an, will einfahren
[  1.0] Auto#1 kommt an, will einfahren
[  1.0] Auto#2 fährt ein
[  1.0] Auto#2 ist drin, parkt
[  1.0] Auto#3 kommt an, will einfahren
[  1.0] Auto#1 fährt ein
[Exception in thread "Thread-0"   1.0] Auto#3 fährt ein
Exception in thread "Thread-2" java.lang.AssertionError
        at CheckedParkhaus.enter(CheckedParkhaus.java)
        at Auto.run(Auto.java)
java.lang.AssertionError
        at CheckedParkhaus.enter(CheckedParkhaus.java)
        at Auto.run(Auto.java)
[  2.0] Auto#2 ist fertig, will ausfahren
[  2.0] Auto#2 fährt aus
Exception in thread "Thread-1" java.lang.AssertionError
```

```
        at CheckedParkhaus.leave(CheckedParkhaus.java)
        at Auto.run(Auto.java)
```

Die Ursache des Problems liegt auf der Hand: Die Methoden `enter` und `leave` werden

- von unterschiedlichen `Auto`-Threads aufgerufen und arbeiten
- mit der*selben* Variablen `autos`.

Sie dürfen dabei nicht unterbrochen werden und müssen unter gegenseitigem Ausschluss ablaufen. Synchronisation von `enter` und `leave` in `CheckedParkhaus` (Listing 6.35) garantiert exklusive Ausführung. `synchronized`-Methoden (Seite 413) sind hier das passende Mittel. Die beiden Methoden kommen sich jetzt gegenseitig nicht mehr ins Gehege.

```
public class SyncedParkhaus implements Parkhaus {
    private final int capacity = 1;

    private volatile int autos = 0;

    public synchronized void enter(Auto auto) throws InterruptedException {
        while(autos == capacity)
            ;
        log(auto + " fährt ein");
        autos++;
        assert autos > 0: "Parkhaus nicht leer";
    }

    public synchronized void leave(Auto auto) {
        log(auto + " fährt aus");
        autos--;
        assert autos < capacity: "Parkhaus hat Platz";
    }

}
```

Listing 6.36: Parkhaus mit synchronisierten Methoden.

Deadlock durch falschen Einsatz von `synchronized`

Allerdings funktioniert das Programm auch mit dieser Maßnahme nicht. Unabhängig von der Anzahl der Prozessoren bleibt es nach kurzer Zeit stecken:

```
$ java -ea SyncedParkhaus 1000/1000 1000/1000 1000/1000
1 processor(s)
[  0.0] Auto#3 fährt los
[  0.0] Auto#1 fährt los
[  0.0] Auto#2 fährt los
```

```
[  1.0] Auto#3 kommt an, will einfahren
[  1.0] Auto#1 kommt an, will einfahren
[  1.0] Auto#2 kommt an, will einfahren
[  1.0] Auto#3 fährt ein
[  1.0] Auto#3 ist drin, parkt
[  2.0] Auto#3 ist fertig, will ausfahren
(keine Reaktion mehr)
```

Gegenseitige Blockade der Threads ohne Ausweg

Das Programm gerät in einen **Deadlock**. In der synchronisierten Implementierung `SyncedParkhaus` (Listing 6.36) haben die Methoden `enter` und `leave` zwar exklusiven Zugriff auf die gemeinsame Variable `autos`, allerdings führt der Zuschnitt der geschützten Codeabschnitte in eine Situation, aus der es keinen Ausweg gibt:

1. Ein Auto kommt am voll besetzten Parkhaus an. Der `Auto`-Thread ruft `enter` auf und wartet dann in der Schleife

   ```
   while(autos == capacity)
      ;
   ```

   darauf, dass ein anderer Thread die Variable `autos` verringert. Dabei befindet er sich in der Methode `enter` und besitzt folglich den Monitor.
2. Ein anderes Auto beendet seine Parkzeit und will das Parkhaus verlassen. Es versucht `leave` aufzurufen, wird aber blockiert, weil der dazu erforderliche Monitor vom ersten Thread belegt ist und nicht zur Verfügung steht.

Beide Threads werden durch eine Anforderung blockiert, die nur der jeweils andere Thread erfüllen kann. Im Beispiel `SyncedParkhaus` (Listing 6.36)

- wird der ersten Thread den Monitor freigeben, *nachdem* der zweite die Variable `autos` verringert.
- wird der zweite Thread die Variable `autos` verringern, *nachdem* der erste den Monitor freigibt.

Einschränkungen zum Vermeiden von Deadlocks

Eine solche gegenseitige Blockade wird als **Deadlock** bezeichnet. Leider kann weder der Compiler noch die JVM vorab auf die Gefahr des Deadlocks hinweisen. Es gibt kein Verfahren, das beliebigen Code untersuchen und darin potenzielle Deadlocks erkennen oder ausschließen könnte. Unter den folgenden Einschränkungen lassen sich Deadlocks aber zuverlässig vermeiden:

1. Kein Thread belegt einen Monitor für unbestimmte Zeit und
2. Threads, die mehrere Monitore brauchen, belegen sie in der gleichen Reihenfolge.

Regelmäßige Freigabe des Monitors

Aufbrechen des synchronisierten Abschnitts

Das Programm `SyncedParkhaus` (Listing 6.36) verstößt gegen die erste Bedingung, weil die Warteschleife den Monitor beliebig lange belegen kann.

Diese Konstruktion lässt sich aufbrechen, wenn die bisherige Methode `enter` zu einer privaten Methode `tryToEnter` modifiziert wird. Die geänderte Fassung testet den Zähler `autos` nur einmal und nicht mehr endlos in einer Schleife. `tryToEnter` zeigt außerdem mit einem `boolean`-Ergebnis an, ob das Auto in das Parkhaus einfahren konnte oder nicht. `enter` selbst ist *nicht synchronisiert* und ruft so lange `tryToEnter` auf, bis diese Erfolg signalisiert:

```
public class TrySyncedParkhaus implements Parkhaus {
    private final int capacity = 1;

    private volatile int autos = 0;

    public void enter(Auto auto) throws InterruptedException {
        while(!tryToEnter(auto))
            ;
    }

    private synchronized boolean tryToEnter(Auto auto) {
        if(autos == capacity)
            return false;
        log(auto + " fährt ein");
        autos++;
        assert autos > 0: "Parkhaus nicht leer";
        return true;
    }

    public synchronized void leave(Auto auto) {
        log(auto + " fährt aus");
        autos--;
        assert autos < capacity: "Parkhaus ist nicht voll";
    }

}
```

Listing 6.37: Regelmäßige Freigabe des Monitors verhindert Deadlock.

Dieses Programm funktioniert zuverlässig:

```
$ java TrySyncedParkhaus 1000/1000 1000/1000 1000/1000
[  0.0] Auto#1 fährt los
[  0.0] Auto#2 fährt los
[  0.0] Auto#3 fährt los
[  1.0] Auto#1 kommt an, will einfahren
[  1.0] Auto#2 kommt an, will einfahren
[  1.0] Auto#3 kommt an, will einfahren
```

```
[  1.0] Auto#1 fährt ein
[  1.0] Auto#1 ist drin, parkt
[  2.0] Auto#1 ist fertig, will ausfahren
[  2.0] Auto#1 fährt aus
[  2.0] Auto#1 ist draußen, fährt weg
[  2.0] Auto#3 fährt ein
[  2.0] Auto#3 ist drin, parkt
[  3.0] Auto#3 ist fertig, will ausfahren
[  3.0] Auto#3 fährt aus
[  3.0] Auto#3 ist draußen, fährt weg
[  3.0] Auto#2 fährt ein
[  3.0] Auto#2 ist drin, parkt
[  4.0] Auto#2 ist fertig, will ausfahren
[  4.0] Auto#2 fährt aus
[  4.0] Auto#2 ist draußen, fährt weg
```

Reihenfolge der Monitor-Belegung

Deadlock beim Belegen mehrerer Monitore

Ein Thread, der verschiedene Datenstrukturen im Zusammenhang manipulieren möchte, braucht gleichzeitigen exklusiven Zugriff. Das ist im Inneren verschachtelter `synchronized`-Blöcke der entsprechenden Monitore gewährleistet:

```
synchronized(monitor1) {
    synchronized(monitor2) {
        synchronized(monitor3) {
            ...
        }
    }
}
```

Das folgende Programm verstößt gegen die zweite der auf Seite 425 genannten Einschränkungen und kann daher in einen Deadlock geraten. Ein `Deadlock`-Objekt speichert zunächst zwei Referenzen in den Objektvariablen `yin` und `yang`. `run` versucht in zwei geschachtelten `synchronized`-Blöcken beide Monitore zu belegen und gibt im Erfolgsfall eine entsprechende Meldung aus:

```
public class Deadlock extends Thread {
    private final Object yin;

    private final Object yang;

    public Deadlock(Object yin, Object yang) {
        this.yin = yin;
        this.yang = yang;
    }

    public void run() {
        synchronized(yin) {
            synchronized(yang) {
```

```
                System.out.println(this + ": yin & yang");
            }
        }
    }

    public static void main(String... args) {
        new Deadlock(1, 2).start();
        new Deadlock(2, 1).start();
    }
}
```

Listing 6.38: Programm mit potenziellem Deadlock.

Die `main`-Methode erzeugt zwei `Deadlock`-Objekte und übergibt ihnen die eindeutigen Wrapper-Objekte `Integer(1)` und `Integer(2)` in unterschiedlicher Reihenfolge. Das Programm zeigt zunächst keine Auffälligkeit:

```
$ java Deadlock
Thread[Thread-0,5,main]: yin & yang
Thread[Thread-1,5,main]: yin & yang
$
```

Das Problem tritt zutage, wenn die beiden Monitore in `run` mit einer Verzögerung belegt werden. Dazu reicht es aus, eine Benutzereingabe einzuschieben:

```
public void run() {
    synchronized(yin) {
        System.console().readLine("Enter to continue ... ");
        synchronized(yang) {
            System.out.println(this + ": yin & yang");
        }
    }
}
```

Listing 6.39: Provokation des Deadlocks.

Bestätigt man die Anfragen der beiden Threads mit der Eingabetaste, dann läuft das Programm in einen Deadlock und kann nur noch von außen abgebrochen werden:

```
$ java Deadlock
Enter to continue ...
Enter to continue ...
```

Ein Thread hat den Monitor `Integer(1)` belegt und wartet auf `Integer(2)`, der andere Thread hat `Integer(2)` belegt und wartet auf `Integer(1)`. Keiner der beiden

Threads gibt „seinen“ Monitor frei und keiner der beiden Threads kann ohne den anderen fortfahren.

Dieser Deadlock lässt sich vermeiden, wenn die Threads die Monitore *in der gleichen Reihenfolge* belegen:

Kein Deadlock bei gleicher Reihenfolge

```
public static void main(String... args) {
    new Deadlock(1, 2).start();
    new Deadlock(1, 2).start();     // vorher: 2, 1
}
```

Listing 6.40: Kein Deadlock bei gleicher Reihenfolge der Monitor-Belegung.

Nachdem beide Threads denselben Monitor zuerst anfordern, kann ihn nur ein Thread erhalten. Dieser Thread erhält in jedem Fall auch den zweiten Monitor, weil ihn kein anderer Thread überhaupt anfordert:

```
$ java Deadlock
Enter to continue ...
Thread[Thread-0,5,main]: yin & yang
Enter to continue ...
Thread[Thread-1,5,main]: yin & yang
$
```

6.5.5 Unveränderliche Klassen

Eine besondere Bedeutung kommt in Zusammenhang mit der Diskussion konkurrierender Zugriffe unveränderlichen Klassen (*immutable class*) zu.[56] Die Objekte unveränderlicher Klassen können, einmal geschaffen, nicht mehr verändert werden. „Änderungen“ lassen sich nur durch Erzeugen neuer, gegenüber dem unveränderlichen Original modifizierter Objekte umsetzen.[57]

Unveränderliche Klassen für sorgenfreie Parallelisierung

Mit der Möglichkeit von Änderungen fallen auch alle Probleme mit konkurrierenden Zugriffen weg. Synchronisation ist schlicht nicht notwendig. Dieser Aspekt ist ein starkes Argument für unveränderliche Klassen.

Allerdings kann längst nicht jede Klasse unveränderlich sein. Beispielsweise ist das `Parkhaus` naturgemäß veränderlich, sodass beim Zugriff durch mehrere Threads kein Weg an Synchronisationsmaßnahmen vorbeiführt.

[56] Unveränderlich sind tatsächlich die Objekte, nicht die Klassen. Bytecode ist in Java grundsätzlich unveränderlich.

[57] Dabei reicht es nicht aus, dass eine Klasse nach außen keine ändernden Methoden anbietet. Sie darf auch keine „internen“ Modifikationen durchführen, beispielsweise um ihre Datenstrukturen zu bereinigen oder zu optimieren.

6.6 Bedingtes Warten

`synchronized` regelt keine Reihenfolge

Die vorhergehenden Abschnitte zeigen, wie sich kritische Operationen mit `synchronized` schützen lassen. Damit ist sichergestellt, dass immer nur ein Thread exklusiven Zugriff auf entsprechende Codeabschnitte hat.

`synchronized` reicht aus, wenn nur der gegenseitige Ausschluss gewährleistet sein muss und ansonsten die Reihenfolge der Abarbeitung kritischer Abschnitte jedem Thread selbst überlassen bleibt.

Schlecht steuern lassen sich dagegen Threads, die kritische Abschnitte in einer bestimmten Reihenfolge durchlaufen, ansonsten aber parallel arbeiten sollen. Dazu sind weitere Sprachmittel nötig, die in diesem Abschnitt vorgestellt werden.

6.6.1 Aktives Warten

Warteschleife frisst unnötig Rechenleistung

Die Parkhaus-Implementierung `TrySyncedParkhaus` (Listing 6.37) arbeitet äußerlich fehlerfrei. Dennoch zeigt sie einen gravierenden Mangel.

Überprüft man während des Programmablaufs mit Systemwerkzeugen die Auslastung, dann stellt sich heraus, dass ein Prozessor durchgehend *mit voller Leistung* läuft, auch wenn das Programm nichts anderes tut, als auf den Ablauf einer Zeitspanne zu warten.

Der Schuldige ist die Methode `enter`, die in einer `while`-Schleife fortwährend `tryToEnter` aufruft, bis schließlich das Ergebnis `true` geliefert wird. Einer der Prozessoren ist dabei mit nichts anderem als der `while`-Schleife beschäftigt.

Aktives versus passives Warten

Diese Art des ständigen Nachfragens wird als „aktives Warten“ (*busy waiting*, *polling*) bezeichnet.[58] Die beiden Methoden `wait` und `notify` machen aktives Warten unnötig und erlauben „passives Warten“.[59] Dazu kommt noch als dritte Methode die Variante `notifyAll` von `notify`.

6.6.2 `wait` und `notify`

`wait` gibt Monitor frei und wartet auf Weckruf

Die Methode `wait` richtet sich an einen Monitor, den der Aufrufer besitzt. Der `wait`-

[58] Man kann sich einen solchen Thread als Hausbewohner vorstellen, der einen Brief erwartet. Bei aktivem Warten läuft der Hausbewohner in einem fort von der Wohnung zum Briefkasten und sieht nach, ob die erwartete Post eingetroffen ist.

[59] Um im Bild des Hausbewohners zu bleiben: Er bleibt in seiner Wohnung und wartet dort geduldig mit `wait`, bis der Postbote nach Einwurf des Briefs mit `notify` die Glocke läutet.

Aufruf gibt den Monitor frei und blockiert dann. Das heißt, dass ein `wait`-Aufruf zunächst nicht zurückkehrt.

`wait` kehrt erst dann zurück, wenn ein anderer Thread

1. mit *demselben* Monitor die Methode `notify` aufgerufen und
2. den Monitor freigegeben hat.

Auch der Aufrufer von `notify` muss den betreffenden Monitor besitzen.

`notify` schickt Weckruf an `wait`-Aufruf

`wait` und `notify` sind Methoden der Klasse `Object` und werden damit von jedem Java-Objekt zur Verfügung gestellt. Typ und Wert des Objekts spielen dabei keine Rolle.

`void wait() throws InterruptedException`
: Wartet auf ein `notify`.

`void notify()`
: Benachrichtigt einen wartenden Thread.

Beide sind als `final` markiert, eine Redefinition ist weder möglich noch sinnvoll. Die Methoden sind nur paarweise und nicht einzeln sinnvoll. Die Aufrufe müssen in zwei verschiedenen Threads stehen, die denselben Monitor benutzen.

`wait` und `notify` richten sich an Monitor

Ein Aufruf von `wait` oder `notify` ohne Besitz des Monitors führt zu einer `IllegalMonitorStateException`. Der Compiler entdeckt diesen Fehler nicht.

Minimalprogramm mit `wait` und `notify`

Das folgende Minimalprogramm zeigt den prototypischen Einsatz von `wait` und `notify`. Es initialisiert die Variable `flag`, startet mit `false` und erwartet am Ende in der Zusicherung den Wert `true`. Das Setzen des `flag` delegiert `main` an den Thread `notifier`. Dessen `run`-Methode wartet auf eine Benutzereingabe und setzt dann `flag`.

```
public class MiniWaitNotify extends Thread {
    private static boolean flag = false;

    public void run() {
        System.console().readLine("Enter to continue ...");
        synchronized(this) {
            flag = true;
            notify();
```

```
        }
    }

    public static void main(String... args) throws InterruptedException {
        Thread notifier = new MiniWaitNotify();
        notifier.start();
        synchronized(notifier) {
            notifier.wait();
        }
        assert flag;
    }
}
```

Listing 6.41: Minimalprogramm mit `wait` und `notify`.

Zusätzliche Ausgabeanweisungen unmittelbar vor und nach den Aufrufen von `wait` und `notify` (im Listing nicht abgedruckt) machen den Ablauf sichtbar:

```
$ java -ea MiniWaitNotify
main: calling wait ...
Enter to continue ...
run: calling notify ...
run: notify called
main: wait returned
$
```

Beide Threads arbeiten mit demselben Monitor. Der `main`-Thread synchronisiert am Thread `notifier` als Monitor und ruft damit `wait` auf. Die `run`-Methode des Threads verwendet `this` als Monitor und richtet den `notify`-Aufruf ebenfalls an das `Thread`-Objekt.

Rückkehr von `wait` nach Wiedererlangen des Monitors

`notify` führt *nicht sofort* zur Rückkehr von `wait` im wartenden Thread. Schließlich belegt der Aufrufer von `notify` noch den Monitor. Würde `wait` tatsächlich mit dem Aufruf von `notify` sofort zurückkehren, dann befänden sich *beide* Threads im `synchronized`-Blocks desselben Monitors. Das darf auf keinen Fall geschehen!

Deshalb muss der aufgeweckte Thread warten, bis der Aufrufer von `notify` seinen `synchronized`-Block verlässt und damit den Monitor freigibt. Erst dann kehrt `wait`, wieder im Besitz des Monitors, zurück. `notify` *merkt* lediglich einen wartenden Thread zur Fortsetzung *vor*, nicht mehr.

`notify` verpufft ohne `wait`

Ein `notify`-Aufruf ist wirkungslos, wenn überhaupt kein Thread wartet. In der folgenden Variante von `MiniWaitNotify` (Listing 6.41) ist der Aufruf von `wait` um drei Sekunden verzögert.

```
public class MiniWaitEarlyNotify extends Thread {
```

```
    private static boolean flag = false;

    public void run() {
        System.console().readLine("Enter to continue ...");
        synchronized(this) {
            flag = true;
            notify();
        }
    }

    public static void main(String... args) throws InterruptedException {
        Thread notifier = new MiniWaitEarlyNotify();
        notifier.start();
        Thread.sleep(3000);
        synchronized(notifier) {
            notifier.wait();
        }
        assert flag;
    }
}
```

Listing 6.42: Verzögertes `wait` mit `notify`, das zu früh ausgelöst werden kann.

Wenn man innerhalb dieser Zeitspanne die Eingabetaste drückt, dann wird `notify` schon *vor* `wait` aufgerufen und „verpufft“ wirkungslos. In diesem Fall wartet `wait` für immer, weil kein weiteres `notify` mehr folgt!

```
$ java -ea MiniWaitEarlyNotify
Enter to continue ...[Enter]    // kurz nach Programmstart
run: calling notify ...
run: notify called
main: calling wait ...
(keine Reaktion mehr)
```

Wenn Sie sich länger als drei Sekunden mit der Eingabetaste Zeit lassen, dann erlöst `notify` den bereits vorher erfolgten `wait`-Aufruf und das Programm endet wie in der ursprünglichen Fassung `MiniWaitNotify` (Listing 6.41):

```
$ java -ea MiniWaitEarlyNotify
Enter to continue ...
main: calling wait ...
[Enter]                         // nach mehr als drei Sekunden
run: calling notify ...
run: notify returned
main: wait returned
```

6.6.3 Test der `wait`-Bedingung

Das vorhergehende Minimalbeispiel zeigt, dass ein `wait`-Aufruf besser testen soll- Test vor `wait`-Aufruf sinnvoll

te, ob die erforderte Bedingung bereits vor dem Aufruf erfüllt ist. In diesem Fall ist der `wait`-Aufruf überhaupt nicht nötig, ja, sogar gefährlich. Im Quelltext sollte `wait` also in der Regel einem `if` untergeordnet sein, das die erwartete Bedingung überprüft, wie im folgenden Programm:

```
public class MiniConditionalWaitNotify extends Thread {
    private static boolean flag = false;

    public void run() {
        System.console().readLine("Enter to continue ...");
        synchronized(this) {
            flag = true;
            notify();
        }
    }

    public static void main(String... args) throws InterruptedException {
        Thread notifier = new MiniConditionalWaitNotify();
        notifier.start();
        Thread.sleep(3000);
        synchronized(notifier) {
            if(!flag) {
                notifier.wait();
            }
        }
        assert flag;
    }
}
```

Listing 6.43: Minimalprogramm mit bedingtem Warten.

Dieses Programm funktioniert unabhängig davon, wann die Eingabetaste gedrückt und damit `notify` aufgerufen wird.

Das Parkhaus-Beispiel lässt sich mit `wait` und `notify` befriedigend lösen. Die Warteschleife in der zwar korrekten, aber ressourcenfressenden Fassung `TrySyncedParkhaus` (Listing 6.37) kann ganz wegfallen. Auch die Hilfsmethode `tryToEnter` ist nicht mehr nötig. Stattdessen ruft die jetzt wieder synchronisierte `enter`-Methode einfach `wait` auf, wenn das Parkhaus voll ist. Im Gegenzug muss `leave` natürlich `notify` aufrufen, um eventuell wartenden Autos den frei gewordenen Platz zu signalisieren:

```
public class WaitingParkhaus implements Parkhaus {
    private final int capacity = 1;

    private volatile int autos = 0;

    public synchronized void enter(Auto auto) throws InterruptedException {
        if(autos == capacity)
```

```
            wait();
        log(auto + " fährt ein");
        autos++;
        assert autos > 0: "Parkhaus nicht leer";
    }

    public synchronized void leave(Auto auto) {
        log(auto + " fährt aus");
        autos--;
        assert autos < capacity: "Parkhaus nicht voll";
        notify();
    }

}
```

Listing 6.44: Mit `wait` und `notify` koordiniertes Parkhaus.

6.6.4 `notify` und `notifyAll`

Im Programm `MiniConditionalWaitNotify` (Listing 6.43) laufen nur zwei Threads, von denen einer `wait` und der andere `notify` aufruft. Im Allgemeinen können aber mehrere Threads am gleichen Monitor warten. `notify` weckt in diesem Fall *irgendeinen* der wartenden Threads auf, während die übrigen weiter im `wait` blockiert bleiben. Der Aufrufer von `notify` kann die Auswahl der JVM nicht beeinflussen.

`notifyAll` weckt alle wartenden Threads auf

Diese Situation findet sich in der Parkhaus-Anwendung `WaitingParkhaus` (Listing 6.44). Dort können mehrere Autos auf die Gelegenheit zur Einfahrt warten. Wenn nur ein Auto das Parkhaus verlässt und dabei `notify` aufruft, kehrt ein beliebiger `wait`-Aufruf der wartenden `Auto`-Threads zurück. Die übrigen erfahren nichts davon und warten weiter. In dieser Anwendung reicht diese zufällige Auswahl aus, aber im Allgemeinen kann davon nicht ausgegangen werden.

In diesen Fällen kommt die `Object`-Methode

```
void notifyAll()
```

zu Hilfe. Im Gegensatz zu `notify` weckt `notifyAll` nicht nur einen, sondern *alle* wartenden Threads auf. Nacheinander kehren die entsprechenden `wait`-Methoden zurück. Es ist dann Sache der betreffenden Threads selbst zu entscheiden, ob sie fortfahren können oder erneut warten müssen. Der Aufrufer von `notifyAll` kann nicht beeinflussen, in welcher Reihenfolge die JVM wartende Threads aufweckt. Die JVM stellt lediglich sicher, dass nacheinander alle an die Reihe kommen.

Aufgeweckte `wait`-Aufrufe kehren nacheinander zurück

Mit `notifyAll` kann eine Ungerechtigkeit im Parkhaus-Beispiel abgestellt werden. In der Fassung `WaitingParkhaus` (Listing 6.44) darf ein *beliebiges* wartendes Auto

einfahren, wenn ein Auto das Parkhaus verlässt. Fairer wäre es dagegen, wenn sich die ankommenden Autos vor dem Parkhaus in einer Schlange einreihen. Wenn ein Platz frei wird, darf das vorderste Auto in der Schlange in das Parkhaus, weil es schon am längsten wartet.

Die neue Klasse erhält als zusätzliche Objektvariable `waiting` eine Liste der wartenden Autos. Ein neu ankommendes Auto wird sofort mit `add` hinten an die Liste angefügt. Sobald es in das Parkhaus einfährt, wird es mit `remove` wieder aus der Liste gelöscht.

```
import java.util.*;

public class FairParkhaus implements Parkhaus {
    private final int capacity = 1;

    private volatile int autos = 0;

    private final List<Auto> waiting = new ArrayList<>();

    public synchronized void enter(Auto auto) throws InterruptedException {
        waiting.add(auto);
        log(auto + " reiht sich ein");
        while(autos == capacity || waiting.indexOf(auto) > 0)
            wait();
        log(auto + " fährt ein");
        waiting.remove(auto);
        autos++;
        assert autos > 0: "Parkhaus ist nicht leer";
    }

    public synchronized void leave(Auto auto) {
        log(auto + " fährt aus");
        autos--;
        assert autos < capacity: "Parkhaus ist nicht voll";
        notifyAll();
    }

}
```

Listing 6.45: Parkhaus mit einer Warteschlange für ankommende Autos.

Aufgeweckte `wait`-Aufrufe entscheiden über Fortsetzung

Anders als in `WaitingParkhaus` (Listing 6.44) ruft `leave` jetzt die Methode `notifyAll` statt `notify` auf. Daraufhin kehren alle `wait`-Aufrufe zurück. Allerdings dürfen keineswegs alle Autos einfahren, sondern nur das vorderste in der Schlange! Der `wait`-Aufruf steht daher in einer `while`-Schleife, die immer wieder `wait` aufruft, solange

1. das Parkhaus voll ist oder
2. das Auto nicht vorne in der Schlange steht.

Obwohl nach einem `notifyAll` alle Autos aufgeweckt werden, fährt nur eines in das Parkhaus ein. Die anderen warten weiter bis zum nächsten `notifyAll`.

Beim Aufruf des Programms mit mehreren Autos, deren Anfahrt zum Parkhaus gleich lang dauert, kann die genaue Reihenfolge des Eintreffens nicht vorhergesagt werden. Allerdings werden die Autos zuverlässig genau in der gleichen Reihenfolge eingelassen, in der sie sich in die Warteschlange einreihen:

```
$ java FairParkhaus 1000/1000 1000/1000 1000/1000
[  0.0] Auto#1 fährt los
[  0.0] Auto#3 fährt los
[  0.0] Auto#2 fährt los
[  1.0] Auto#1 kommt an, will einfahren
[  1.0] Auto#2 kommt an, will einfahren
[  1.0] Auto#3 kommt an, will einfahren
[  1.0] Auto#1 reiht sich ein
[  1.0] Auto#1 fährt ein
[  1.0] Auto#1 ist drin, parkt
[  1.0] Auto#3 reiht sich ein
[  1.0] Auto#2 reiht sich ein
[  2.0] Auto#1 ist fertig, will ausfahren
[  2.0] Auto#1 fährt aus
[  2.0] Auto#1 ist draußen, fährt weg
[  2.0] Auto#3 fährt ein
[  2.0] Auto#3 ist drin, parkt
[  3.0] Auto#3 ist fertig, will ausfahren
[  3.0] Auto#3 fährt aus
[  3.0] Auto#3 ist draußen, fährt weg
[  3.0] Auto#2 fährt ein
[  3.0] Auto#2 ist drin, parkt
[  4.0] Auto#2 ist fertig, will ausfahren
[  4.0] Auto#2 fährt aus
[  4.0] Auto#2 ist draußen, fährt weg
```

Zusammenfassung

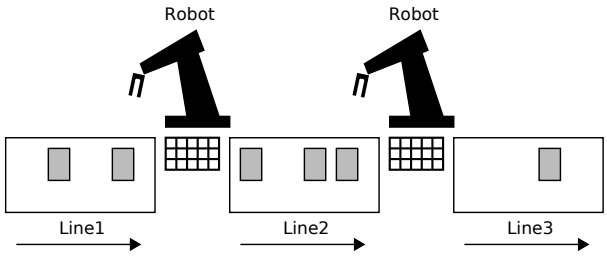
- Die Methode `start` der Klasse `Thread` führt zum **parallelen Aufruf** der Methode `run` durch die JVM. `run` wird in abgeleiteten Klassen mit dem gewünschten Verhalten redefiniert.
- Die genauen **Zeitverhältnisse** parallel laufender Threads sind **nicht deterministisch**.
- Threads können nur **einmal** gestartet werden. Sie enden mit der Rückkehr aus `run`.
- Das **Interface** `Runnable` bietet eine **Alternative** zur Ableitung der Basisklasse `Thread`.
- Ein Java-Programm endet, wenn alle Threads beendet sind. Davon ausgenommen sind **Daemon-Threads**.

- **Interrupts** erlauben eine einfache Kommunikation zwischen Threads.
- Die JVM verwaltet Threads in einem **Lebenszyklus** aus mehreren Zuständen, der nur mittelbar beeinflusst werden kann.
- Die JVM verteilt Threads auf **Prozessoren**. Wenn die Prozessoren nicht ausreichen, kommen die Threads reihum jeweils für kurze Zeit an die Reihe.
- Jeder Thread hat eigene lokale Variablen, aber die Objekte teilen sich alle. Bei **gleichzeitigem Zugriff** kommt es zu Fehlern.
- **Synchronisation** stellt sicher, dass immer nur *ein einziger* Thread einen Codeabschnitt, der auf geteilte Daten zugreift, **exklusiv ausführt**.
- Korrekte Synchronisation ist schwierig. Programme können in **Deadlocks** stecken bleiben oder bei **Übersynchronisierung** den Nutzen von Threads ganz verlieren.
- Mit `wait` kann ein Thread ohne Verbrauch von Rechenleistung darauf **warten,** dass ihm ein anderer Thread mit `notify` signalisiert, dass die **Bedingungen** zur Fortsetzung **erfüllt** sind.
- Mit `wait` und `notify` lassen sich beispielsweise **Erzeuger-Verbraucher-Systeme** effizient implementieren.

Aufgaben

Aufgabe 1: Roboter am Fließband

In einer modernen Fabrik fertigen Industrieroboter Bauteile. Ein Roboter nimmt ein Bauteil von einem Fließband auf, bearbeitet es und legt es dann auf einem anderen Fließband ab, an dessen Ende der nächste Roboter wartet. Die Roboter wiederholen den Zyklus Aufnehmen, Bearbeiten, Ablegen so lange, bis sie gestoppt werden. Die folgende Skizze zeigt drei Fließbänder mit zwei Robotern:



Die beiden Klassen `AssemblyLine` und `Robot` repäsentieren ein Fließband und einen Industrieroboter.

Die Variable `parts` in `AssemblyLine` speichert die Anzahl der Bauteile, die momentan daraufliegen. Es hat beliebig viel Platz. Der Konstruktor erwartet die Anzahl Bauteile, die bereits zu Beginn auf den Band bereitliegen. Mit den Methoden `putdown` und `pickup` legt ein Roboter ein Bauteil ab oder nimmt eines auf. Aufrufe der statischen Hilfsmethode `log` protokollieren diese Aktionen auf dem Bildschirm.[60]

```
public class AssemblyLine {
    private int parts;

    public AssemblyLine(int initialParts) {
        parts = initialParts;
    }

    public  void putdown(Robot robot) {
        parts++;
        log(robot, "ist fertig, legt Bauteil ab\t(" + this + ")");
    }

    public void pickup(Robot robot) throws InterruptedException {
        parts--;
        log(robot, "nimmt Bauteil, arbeitet\t(" + this + ")");
    }

    public int getParts() {
        return parts;
    }
}
```

Listing 6.46: Fließband mit Bauteilen, die Roboter darauf aufnehmen und ablegen.

Die Roboter arbeitet alle gleichzeitig und werden deshalb mit einer `Thread`-Klasse modelliert. Die Methode `run` durchläuft in einer Endlosschleife den Arbeitszyklus des Roboters. Ein Roboter wird mit dem liefernden (`lineIn`) und dem abnehmenden Fließband (`lineOut`) sowie der Dauer seines Arbeitsschritts (`workTime`) initialisiert.[61]

```
public class Robot extends Thread {
    private final AssemblyLine lineIn;

    private final AssemblyLine lineOut;

    private final int workTime;

    public Robot(AssemblyLine lineIn, AssemblyLine lineOut, int workTime) {
```

[60] Eine einfache `toString`-Methode, die die Identität und die Anzahl der Bauteile benennt, ist nicht abgedruckt. Die Exceptionsignatur von `pickup` wird später gebraucht.

[61] Die einzeiligen Getter und eine `toString`-Methode sind nicht abgedruckt.

```
        this.lineIn = lineIn;
        this.lineOut = lineOut;
        this.workTime = workTime;
    }

    public void run() {
        try {
            while(true) {
                lineIn.pickup(this);
                Thread.sleep(workTime);
                lineOut.putdown(this);
            }
        }
        catch(InterruptedException ie) {
            throw new AssertionError(ie);
        }
    }

}
```

Listing 6.47: Industrieroboter, der Bauteile von einem Fließband aufnimmt, bearbeitet und auf dem nächsten Fließband ablegt.

Das folgende Hauptprogramm modelliert den Aufbau, der in der Skizze auf Seite 438 gezeigt ist. Zunächst werden drei `AssemblyLines` definiert, davon das erste mit zehn Bauteilen und die beiden anderen leer. Das Array `robots` nimmt die beiden Roboter auf, von denen der erste langsamer arbeitet als der zweite. Dann startet das Programm die Roboter.

```
public class AssemblyLineMain {
    public static void main(String... args) throws InterruptedException {
        AssemblyLine line1 = new SyncedAssemblyLine(10);
        AssemblyLine line2 = new SyncedAssemblyLine(0);
        AssemblyLine line3 = new SyncedAssemblyLine(0);
        Robot[] robots = new Robot[] {
            new InterruptibleRobot(line1, line2, 1000),
            new InterruptibleRobot(line2, line3, 400),
        };
        for(Robot robot: robots)
            robot.start();
    }

}
```

Listing 6.48: Hauptprogramm, das eine Fließbandkette mit Robotern simuliert.

Ein Programmstart zeigt sofort einen sinnlosen Verlauf, weil der Bauteilzähler ins Negative gerät:

```
$ java AssemblyLineMain
```

```
[  0.0] Robot#2: nimmt Bauteil, arbeitet        (Band 2: -1 Bauteil(e))
[  0.0] Robot#1: nimmt Bauteil, arbeitet        (Band 1: 9 Bauteil(e))
[  0.5] Robot#2: ist fertig, legt Bauteil ab    (Band 3: 1 Bauteil(e))
[  0.5] Robot#2: nimmt Bauteil, arbeitet        (Band 2: -2 Bauteil(e))
...
```

Das Programm endet nicht alleine und muss abgebrochen werden.

Wartende Roboter

Ein Roboter kann nicht weiterarbeiten, wenn das liefernde Fließband leer ist. Leiten Sie von `AssemblyLine` eine neue Klasse `SyncedAssemblyLine` ab. Die redefinierte Methode `pickup` blockiert bei einem leeren Band so lange, bis wieder ein Bauteil daraufliegt. Dieser Umstand ist in `putdown` sichergestellt und kann dort signalisiert werden. Ändern und kopieren Sie keinen bestehenden Code, abgesehen von den Konstruktoraufrufen in `AssemblyLineMain`.

Jetzt sollten die beiden Roboter im Takt arbeiten:

```
$ java AssemblyLineMain
[  0.0] Robot#1: nimmt Bauteil, arbeitet        (Band 1: 9 Bauteil(e))
[  0.0] Robot#2: wartet, Band leer
[  1.0] Robot#1: ist fertig, legt Bauteil ab    (Band 2: 1 Bauteil(e))
[  1.0] Robot#1: nimmt Bauteil, arbeitet        (Band 1: 8 Bauteil(e))
[  1.0] Robot#2: nimmt Bauteil, arbeitet        (Band 2: 0 Bauteil(e))
[  1.5] Robot#2: ist fertig, legt Bauteil ab    (Band 3: 1 Bauteil(e))
[  1.5] Robot#2: wartet, Band leer
...
```

Beobachten Sie das Verhalten des Programms bei Variationen der Anordnung:

- Lassen Sie den zweiten Roboter langsamer arbeiten als den ersten.
- Verlängern Sie die Kette um weitere Fließbänder und Roboter.
- Geben Sie dem ersten Roboter einen Kollegen gleicher Bauart, der parallel an den gleichen Fließbändern arbeitet.
- Geben Sie dem zweiten Roboter zwei unterschiedlich schnelle Kollegen.
- Konstruieren Sie einen Fließband-Zyklus. Das ist zwar etwas wirklichkeitsfremd, muss aber dennoch geordnet funktionieren.

Programmende

Die synchronisierten Fließbänder führen zu einer realistischen Simulation, deren Ablauf im Vorhinein nicht unbedingt klar ist. Allerdings bleiben die Roboter und

das Programm stehen, wenn am Ende alle Bauteile auf dem letzten Fließband liegen.

Erweitern Sie die `main`-Methode um eine Überwachung des letzten Fließbands. Sobald sich dort alle Bauteile angesammelt haben, die ursprünglich auf dem ersten Fließband lagen, schickt `main` allen Robotern einen Interrupt. Die Klasse `Robot` (Listing 6.47) ignoriert die Interrupts allerdings, sodass die Maßnahme zunächst keine Wirkung hat.

Leiten Sie von `Robot` eine neue Klasse `InterruptibleRobot` ab, die bei einem Interrupt aus der `run`-Methode zurückkehrt.

Jetzt endet das Programm regulär, wenn das letzte Bauteil fertiggestellt ist.

Kontrollieren Sie während des Programmlaufs die Auslastung Ihres Systems. Bei korrekten Einsatz der Sprachmittel von Java darf sich keine nennenswerte Prozessorlast zeigen, weil die Roboter ihre Arbeit nur mit `sleep`-Aufrufen simulieren.

Aufgabe 2: Buchstaben-Ticker

Das folgende Programm wartet auf Tastatureingaben. Bei einem kleinen Vokal startet es eine Schleife, die den entsprechenden Großbuchstaben zehnmal nacheinander im Abstand von jeweils einer Sekunde ausgibt. Ein `q` beendet das Programm.

```
import java.io.*;

public class SerialLetterTicker {
    public static void main(String... args) throws IOException, InterruptedException {
        int chr = 0;

        while(chr != 'q') {
            chr = System.in.read();
            switch(chr) {
                case 'a':
                case 'e':
                case 'i':
                case 'o':
                case 'u':
                    for(int n = 0; n < 10; n++) {
                        System.out.print((char)Character.toUpperCase(chr));
                        System.out.flush();
                        Thread.sleep(1000);
                    }
                    break;
            }
        }
    }
```

```
}
```

Listing 6.49: Programm, das auf die Eingabe kleiner Buchstaben wartet und dann Großbuchstaben ausgibt.

Denken Sie daran, dass Sie immer die Eingabetaste drücken müssen, damit Tastatureingaben auch im Programm ankommen. Auf die Arbeitsweise des Programms hat das aber keinen Einfluss.[62] Das Programm arbeitet sequenziell. Wenn Sie zwei Buchstaben zusammen eingeben, erscheinen die Ausgaben nacheinander:

```
$ java SerialLetterTicker
a
AAAAAAAAAAe
EEEEEEEEEEou
OOOOOOOOOOUUUUUUUUUUq
$
```

Schreiben Sie ein neues Programm `ParallelLetterTicker`, das bei Ausgaben nicht blockiert. Es akzeptiert also *jederzeit* Eingaben und und verarbeitet sie sofort, auch während andere Ausgaben laufen. Die Ausgaben mehrerer Eingaben mischen sich jetzt und erscheinen, jede für sich, im korrekten Zeittakt.

```
$ java ParallelLetterTicker
a
AAAAe
EAEAEAEAEAEAEEEEou
OUOUOUOUUOUOUOUOUOUq
```

Verschieben Sie dazu die Ausgabeschleife in eine getrennte `Thread`-Klasse `LetterPrinter`. Die Eingabe eines kleinen Buchstabens erzeugt ein `LetterPrinter`-Objekt mit den passenden Argumenten und startet den Thread.

Synchronisation

Erweitern Sie das Programm so, dass die Eingabe eines `b` alle laufenden Ausgaben für die Dauer von fünf Sekunden unterdrückt. Wohlgemerkt: Die Ausgaben laufen im Hintergrund weiter, nur erscheinen sie nicht auf dem Bildschirm.

[62] In einem Unix-Terminal schaltet das Kommando `stty raw` die Eingabepufferung ab. Tastendrücke kommen dann *sofort* beim Programm an. Allerdings sollte das Terminal anschließend mit `stty sane` wieder in den normalen Betriebszustand versetzt werden. Die Eingabe „`stty raw; java SerialLetterTicker; stty sane`“ führt nur das Java-Programm im entsprechenden Modus aus und bringt das Terminal nachher wieder in einen brauchbaren Zustand.

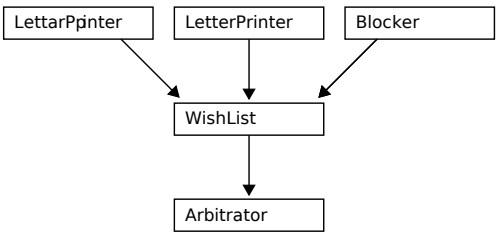
Die harmlos wirkende neue Anforderung löst einige Änderungen aus. Die `LetterPrinter`-Threads dürfen jetzt nicht mehr selbst Buchstaben ausgeben, sondern können nur noch *Wünsche* äußern. Dazu schreiben die `LetterPrinter` ihre Wünsche in eine Wunschliste.

Ob und welche Wünsche erfüllt werden, entscheidet ein sogenannter `Arbitrator`. Er arbeitet immer wieder die Wunschliste durch und sieht sich die Wünsche an. Einen Teil davon führt er aus, andere wirft er weg.

Auch die Eingabe eines `b` startet nun einen Thread der Klasse `Blocker`. Ein solcher Thread schreibt den Wunsch „keine Ausgaben“ in die Wunschliste. Nach Ablauf der vereinbarten Zeitspanne nimmt der `Blocker` den Wunsch wieder zurück und endet dann.

Offen bleibt die Frage, nach welchen Kriterien der `Arbitrator` Wünsche erfüllt oder verwirft. Dazu erhält jeder Wunsch eine ganzzahlige Priorität. Buchstaben der `LetterPrinter` haben die Priorität 1, die Bitte um Stille der `Blocker` dagegen die höhere Priorität 2. Der `Arbitrator` führt einfach die Wünsche mit der momentan höchsten Priorität aus und wirft die anderen weg.

Diese Konstruktion zeigt eine typische Erzeuger-Verbraucher-Struktur. Die `LetterPrinter` und `Blocker` „produzieren“ Wünsche, der `Arbitrator` „verbraucht“ sie. Die Wunschliste ist der gemeinsame Puffer, über den Erzeuger und Verbraucher miteinander kommunizieren. Daraus ergibt sich, dass alle Zugriffe auf die Wunschliste synchronisiert sein müssen.



Ein Punkt verdient noch einen Hinweis: Die Wünsche der `LetterPrinter` bearbeitet der `Arbitrator` sofort, ob er sie nun erfüllt oder nicht. Diese Wünsche sind damit erledigt und der `Arbitrator` kann sie aus der Liste löschen. Die Wünsche der `Blocker` müssen dagegen in der Liste bleiben, bis der `Blocker` selbst sie aus der Liste entfernt und damit zurücknimmt. Die Klasse der Wünsche braucht also ein drittes Attribut „sticky“ (abgesehen vom Text und von der Priorität), das dem `Arbitrator` sagt, ob er den betreffenden Wunsch in der Liste lassen soll oder daraus löschen kann.

Insgesamt umfasst Ihre Lösung die folgenden Klassen:

`ParallelLetterTicker`
: Hauptprogramm, das die Standardeingabe liest und passende Threads startet.

`Wish`
: Ein Wunsch mit Text, Priorität und „sticky“-Flag.

`WishList`
: Die Wunschliste mit Wünschen als Elemente. Alle Zugriffe sind synchronisiert.

`Arbitrator`
: Ein Thread, der die Wunschliste liest und die Wünsche erfüllt oder ignoriert.

`LetterPrinter`
: Threads, die in regelmäßigem Rhythmus Wünsche zur Ausgabe von Buchstaben in die Wunschliste schreiben.

`Blocker`
: Threads, die Wünsche niedrigerer Priorität verhindern.

Erweiterung

Die bisher entwickelte Struktur erweist sich als modular und flexibel.[63] Beispielsweise sollten sich ohne Änderungen des Codes mehrere `Blocker` überlagern können.

Testen Sie die Belastbarkeit der Konstruktion mit zwei Erweiterungen, die mit geringem Aufwand umsetzbar sein sollten:

1. Die Eingabe `x` löst eine „Eilmeldung“ aus, die sofort den Text „Attention!“ ausgibt und dann nach einer halben Sekunde noch den Text „Emergency!“. `Blocker` spielen dabei keine Rolle.
2. Die Eingabe `f` startet einen Flut von 100 Punkten, die im Abstand von einer zehntel Sekunde erscheinen. Diese Flut lässt sich nicht blockieren und unterdrückt alle anderen Ausgaben, abgesehen von Eilmeldungen.

Überprüfen Sie mit geeigneten Werkzeugen die Auslastung des Systems, während das Programm läuft. Bei korrekter Implementierung sollte sich keine nennenswerte Prozessorlast zeigen, was immer das Programm gerade tut.

[63] Diese Konstruktion spiegelt einen wesentlichen Teil der „Subsumption-Architektur“ wider, die in der Robotik eine Rolle spielt.

Kapitel

7 Netzwerkprogrammierung

Lernziele

In diesem Kapitel lernen Sie

- die **Grundlagen** von Netzwerken kennen, die gleichermaßen zur Netzwerkprogrammierung in Java und anderen Programmiersprachen notwendig sind.
- wie Sie **Clients und Server** schreiben, die in einem Netzwerk Dienste anbieten und nutzen können.
- wie HTML (*Hypertext Markup Language*) und HTTP (*Hypertext Transfer Protocol*) funktionieren und wie Sie mit Java **Webserver** entwickeln, die mit beliebigen Browsern benutzt werden können.

Heute ist praktisch jeder Rechner an ein Netzwerk angebunden und kommuniziert mehr oder weniger offensichtlich mit anderen Computersystemen. Zum Teil verwischen sich die Grenzen zwischen den eigenen lokalen Ressourcen und den Ressourcen auf entfernten Systemen so sehr, dass der Unterschied oft nicht mehr klar erkennbar ist. Ob das ein Segen oder ein Fluch ist, muss sich noch erweisen.

Am Datenaustausch sind sowohl Hardware- wie auch Softwarekomponenten beteiligt. In diesem Kapitel geht es um Letzteres, genauer gesagt, um die Mittel und Wege, wie Java-Programme über ein Netzwerk mit anderen Programmen zusammenarbeiten können. Diese „anderen Programme“ können selbst wieder Java-Programme sein, aber auch Programme in einer ganz anderen Implementierungssprache.

7.1 Grundlagen

7.1.1 ISO/OSI-Schichtenmodell

Einteilung von Netzwerksoftware in Ebenen

Netzwerkkommunikation kann aus verschiedenen Blickwinkeln betrachtet wer-

den. So ist beispielsweise der Download einer Datei aus dem Internet eine Sicht auf Netzwerkkommunikation, ebenso wie die Lichtimpulse in einer Glasfaserleitung neben der Autobahn. Das „ISO/OSI-Schichtenmodell“ ordnet diese unterschiedlichen Sichten.

Das Modell teilt die Kommunikation in sieben Schichten ein. Jede Schicht, abgesehen von der untersten, nutzt die Dienste der darunterliegenden Schicht und stellt außerdem selbst Leistungen für die darüberliegende Schicht zur Verfügung. Für die Netzwerkprogrammierung mit Java sind nicht alle Schichten gleichermaßen relevant. Die folgende Liste gibt einen Überblick:

Jede Ebene bietet Dienste für höhere Ebene

1. Physikalische Schicht (*physical layer*, unterste Schicht)
   Stellt die Mittel zur Verfügung, um zwischen zwei Systemen Bits zu übertragen. Dabei spielt es keine Rolle, ob die Bits als elektrische Signale, optische Impulse, elektromagnetische Wellen oder in einer anderen Form transportiert werden.

2. Sicherungsschicht (*link layer*)
   Überträgt Bit-Pakete (*frame*), die mit Prüfsummen versehen sind und so gegebenenfalls als fehlerhaft erkannt werden können. Ein typisches Protokoll auf dieser Ebene ist Ethernet. Die Kommunikationspartner sind hier noch direkt verbunden und identifizieren sich mit „Hardware-Adressen“ (*Media Access Control Address*, MAC-Adresse).

3. Vermittlungsschicht (*network layer*)
   Vermittelt Datenpakete über verschiedene Netzwerke hinweg (*routing*) und sorgt dabei dafür, dass jedes Datenpaket seinen Weg findet. Protokolle sind beispielsweise IP für den eigentlichen Datentransport und ICMP (*Internet Control Message Protocol*) für Statusabfragen. Teilnehmer verfügen über logische „IP-Adressen“.

4. Transportschicht (*transport layer*)
   Kann Daten sicher zustellen und korrigiert Fehler, beispielsweise durch Nachsenden fehlender Pakete. Bekannte Protokolle sind TCP und UDP. TCP (*Transmission Control Protocol*) ist auf Zuverlässigkeit ausgelegt. UDP (*User Datagram Protocol*) arbeitet schneller, aber auch etwas „sorgloser“ und eignet sich zum Beispiel für Liveübertragungen.

5. Sitzungsschicht (*session layer*)
   Regelt komplette Dialogsequenzen zwischen Anwendungen, vom Verbindungsaufbau bis zum Abbau. Diese und die höheren Schichten betreffen nicht mehr allgemeinen Datentransport, sondern orientieren sich an bestimmten Anwendungen.

6. Darstellungsschicht (*presentation layer*)
   Legt bestimmte Datenformate fest, mit denen Anwendungen Informationen austauschen. Hier wird beispielsweise das Encoding von Textdaten oder der Einsatz von Kompressions- und Verschlüsselungsalgorithmen ausgehandelt.

7. Anwendungsschicht (*application layer*)
   Regelt die Kommunikationsverfahren einzelner Anwendungen, wie zum Beispiel FTP (*File Transfer Protocol*), E-Mail und Webservices.

Untere Schichten klar abgegrenzt

Über die konkreten Aufgaben der Schichten 1 bis 4 herrscht weitgehend Einigkeit. Der Zweck der Schichten 5 bis 7 ist zwar im ISO/OSI-Standard beschrieben, in der Praxis gibt es aber einen gewissen Auslegungsspielraum. Die Bedeutung des Standards liegt daher in den unteren Ebenen.

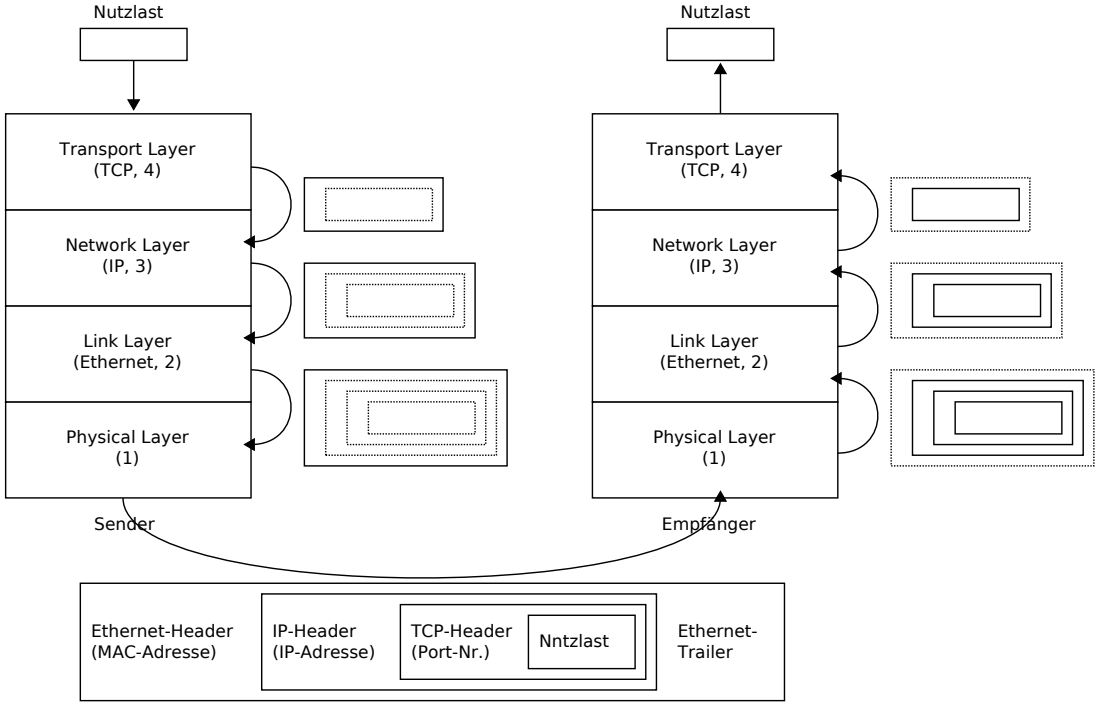
Für die Programmierung von Netzwerkanwendungen mit Java sind in erster Linie die Schichten 3 und 4 interessant. Um die Kommunikation auf Ebene der Schichten 1 und 2 kümmern sich das Betriebssystem und die virtuelle Maschine. Den höheren Ebenen entsprechen einzelne konkrete Anwendungen, wie beispielsweise Webanwendungen (Abschnitt 7.3).

7.1.2 Protokollstapel

Daten durchlaufen Ebenen beim Sender und beim Empfänger

Die Kommunikation auf jeder Ebene des ISO/OSI-Schichtenmodells ist durch Protokolle geregelt, die den zeitlichen Ablauf und die Interpretation der übermittelten Daten genau festlegen. Alle Protokolle zusammen ergeben einen Stapel von Protokollen, den **Protokollstapel** (*protocol stack*).

Angenommen, ein Programm auf einem Rechner („Sender“) möchte Daten an einen anderen Rechner („Empfänger“) schicken. Diese „Nutzlast“ durchläuft auf Senderseite den Protokollstapel von oben nach unten. Jede Schicht versieht die Daten der darüberliegenden Ebene mit eigenen Verwaltungsinformationen und reicht das erweiterte Paket an die darunterliegende Schicht weiter.



Verwaltungsdaten kapseln Nutzlast

Auf Schicht 1 geht das gesamte Datenpaket mit allen Headern und Trailern auf die Reise und trifft irgendwann auf Ebene 1 des Protokollstapels des Empfängers ein. Dort durchläuft es den gleich aufgebauten Protokollstapel von unten nach oben. Jede Ebene trennt „ihre“ Verwaltungsdaten vom Paket ab, wertet sie aus und reicht den Rest an die darüberliegende Ebene weiter.

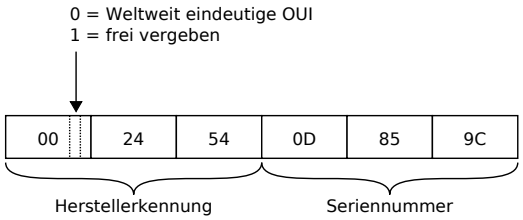
Schließlich erreicht die Nutzlast das Zielprogramm auf dem Empfängersystem.

7.1.3 MAC-Adressen (Ethernet)

Hardware-Adressen identifizieren Netzwerkschnittstellen

Auf Ebene 2 identifizieren sich Systeme mit „Hardware-Adressen“ (*Media Access Control Address*, MAC-Adresse). MAC-Adressen sind weltweit eindeutig für jede Netzwerkschnittstelle eines Rechners. Logisch sind MAC-Adressen 48-stellige Binärzahlen, wobei die ersten 24 Bits den Hersteller codieren (*Organizationally Unique Identifier*, OUI) und die zweiten 24 Bits eine Seriennummer des Herstellers.[1]

[1] In Wahrheit liegen die Verhältnisse komplizierter: Bit 2 des ersten Bytes entscheidet darüber, ob die ersten 24 Bits einen OUI repräsentieren (0) oder frei vergeben sind (1).



Das folgende Programm gibt die MAC-Adresse der Netzwerkschnittstelle aus, die auf der Kommandozeile angegeben ist:

Auslesen der MAC-Adresse

```
import java.net.*;

public class MACAddress {
    public static void main(String... args) throws SocketException {
        NetworkInterface networkInterface = NetworkInterface.getByName(args[0]);
        byte[] macAddress = networkInterface.getHardwareAddress();

        // Ausgabe
        for(byte b: macAddress)
            System.out.printf("%02X ", b);
        System.out.println();
    }
}
```

Listing 7.1: MAC-Adresse der Netzwerkschnittstelle laut Kommandozeile ausgeben.

Auf dem System des Autors ergibt sich:

```
$ java MACAddress eth0
00 24 54 0D 86 9C
```

Die ersten 3 Bytes dieses Beispiels (`00 24 54`) sind dem koreanischen Hersteller „Samsung Electronics Co., LTD“ zugeordnet. Die weiteren 3 Bytes (`0D 86 9C`) unterliegen der Kontrolle des Herstellers.[2]

Ein Rechner kann über mehrere Netzwerkschnittstellen verfügen, von denen jede eine eigene MAC-Adresse hat. Das folgende Programm listet die Netzwerkschnittstellen auf:

Liste der Netzwerkschnittstellen eines Systems

```
import java.net.*;
import java.util.*;
```

[2] Der Dienst `http://hwaddress.com` hilft beim Ermitteln des Herstellers einer MAC-Adresse.

```
public class ListNetworkInterfaces {
    public static void main(String... args) throws SocketException {
        for(Enumeration<NetworkInterface> interfaces = NetworkInterface.getNetworkInterfa
                interfaces.hasMoreElements();)
            System.out.println(interfaces.nextElement().getDisplayName());
    }
}
```

Listing 7.2: Netzwerkschnittstellen auflisten.

Das Programm gibt beispielsweise aus:

```
$ java ListInterfaces
eth0
lo
tap0
wlan0
```

Diese Namen sind systemspezifisch. Die Schnittstelle „lo“ spielt eine besondere Rolle, die im nächsten Abschnitt diskutiert wird.

Hardware-Adressen lassen sich mit geeigneten Hilfsmitteln und Zugriffsrechten ändern. Sie sind also nicht so verlässlich, wie man vermuten könnte.[3]

7.1.4 Hostadressen (IP)

IP-Adressen identifizieren einen Host

Während MAC-Adressen (siehe vorhergehender Abschnitt) an Hardware gebunden sind, arbeiten die höheren Schichten der Netzwerkkommunikation mit logischen Adressen, den sogenannten **IP-Adressen**.[4] IP-Adressen gibt es heute in zwei Formaten, als IPv4- (32-stellige Binärzahlen) und als IPv6-Adressen (128-stellige Binärzahlen). IPv6 wird auf lange Sicht IPv4 verdrängen, jedoch bremst der nicht unerhebliche Aufwand zur Umstellung den Umstieg.[5]

Aufteilung in Netzwerk- und Hostadresse

Die übliche Schreibweise teilt die 32 Bits einer IP-Adresse in vier einzelne Bytes

[3] Eine Internetsuche nach den Begriffen „MAC“ und „change“ liefert schnell passende Software für Ihr System.

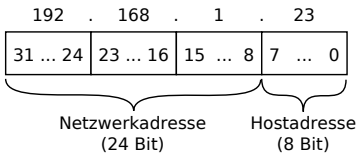
[4] Neben dem „Internet Protocol“ (IP) gibt es auf Ebene 2 auch andere Protokolle wie zum Beispiel ICMP. IP-Adressen sind streng genommen spezifisch für das IP und darauf aufbauende Protokolle, über das aber heute der überwiegende Teil der Kommunikation abläuft.

[5] Dieser Text verwendet das heute weiter verbreitete IPv4. Für die Themen dieses Kapitels spielt der Unterschied zwischen IPv4 und IPv6 keine große Rolle.

auf, deren dezimale Werte mit Punkten getrennt aneinandergesetzt werden, wie zum Beispiel:[6]

```
192.168.1.23
```

Die 32 Bits einer IP-Adresse sind aufgeteilt in einen Vorspann für die **Netzwerkadresse** (höherwertige Bits) und einen Abspann für die **Hostadresse** (restliche niederwertige Bits).



Die Grenze zwischen der Netzwerkadresse und der Hostadresse liegt nicht fest, sondern kann innerhalb der 32 Bits nach Bedarf verschoben werden. In der Schreibweise für IP-Adressen legt eine mit / angefügte Zahl fest, wie viele führende Bits zur Netzwerkadresse gehören. Der Rest entfällt automatisch auf die Hostadresse. Beispielsweise adressiert

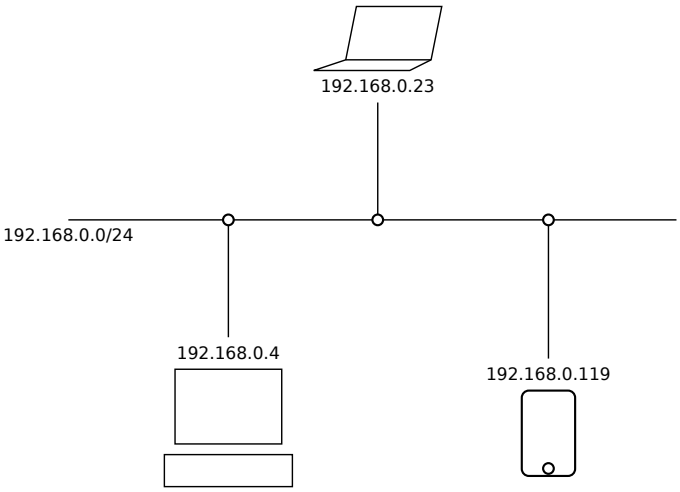
```
192.168.1.0/24
```

ein Netzwerk mit der 24 Bit breiten Netzwerkadresse `192.168.1`. Von den insgesamt 32 Bits der IP-Adresse bleiben nach Abzug der 24 Bits für die Netzwerkadresse noch 8 Bits übrig. In diesem Netzwerk stehen also 2^8 = 256 Hostadressen zur Verfügung.

Hosts eines Netzwerks direkt verbunden

Ein „Netzwerk“ umfasst alle Rechner, die direkt verbunden sind und ohne Vermittler miteinander kommunizieren. Sie haben alle die gleiche Netzwerkadresse, aber unterschiedliche Hostadressen. Hostadressen müssen nicht fortlaufend vergeben werden, sondern können beliebig verteilt sein.

[6] Diese Schreibweise ist verbreitet, aber nicht die einzige. Gleichwertig sind zum Beispiel eine einzelne Hexadezimalzahl (`C0A80117`) oder vorzeichenlose Dezimalzahl (`3232235799`). Viele Programme kommen mit unterschiedlichen Schreibweisen zurecht.



Besondere Hostadressen

Zwei Hostadressen innerhalb eines Netzwerks sind besonderen Zwecken vorbehalten:

- Alle Bits 0: adressiert das Netzwerk selbst und keinen bestimmten Host.
- Alle Bits 1: spricht *jeden* Host im Netzwerk an (*Broadcast*).

Im Netzwerk `192.168.1.0/24` gibt es also höchstens 254 Hosts mit Adressen von `192.168.1.1` bis `192.168.1.254`.

Liste der IP-Adressen und Netzwerkschnittstellen

Es gibt keine eindeutige Zuordnung zwischen Netzwerkschnittstellen und IP-Adressen. Eine Netzwerkschnittstelle kann unter mehreren verschiedenen IP-Adressen erreichbar sein. Das folgende Programm listet alle Netzwerkschnittstellen und deren IP-Adressen auf. Die Angabe in Klammern gibt die Anzahl der Bits an, die auf die Netzwerkadresse entfallen:

```
import java.net.*;
import java.util.*;

public class IPAddresses {
    public static void main(String... args) throws SocketException {
        for(Enumeration<NetworkInterface> interfaces =
                    NetworkInterface.getNetworkInterfaces();
                interfaces.hasMoreElements();) {
            NetworkInterface networkInterface = interfaces.nextElement();
            for(InterfaceAddress address: networkInterface.getInterfaceAddresses())
                System.out.printf("%s\t%s (%d)%n",
                                  networkInterface.getDisplayName(),
                                  address.getAddress(),
                                  address.getNetworkPrefixLength());
        }
```

```
    }
}
```

Listing 7.3: Netzwerkschnittstellen mit IP-Adressen und Anzahl Bits der Netzwerkadresse auflisten.

Es liefert beispielsweise die folgende Ausgabe:[7]

```
$ java IPAddresses
eth0    /129.187.208.11 (24)
lo      /127.0.0.1 (0)
tap0    /10.4.0.1 (24)
wlan0   /192.168.1.23 (24)
```

IP-Adressen und Hostnamen global eindeutig

IP-Adressen und symbolische Hostnamen (siehe Seite 457) sind global eindeutig. Für eine geordnete Zuteilung sorgt die Non-Profit-Organisation ICANN (*Internet Corporation for Assigned Names and Numbers*) mit Sitz in den USA.[8] Die ICANN delegiert die Aufgaben zum Teil an lokale Organisationen in einzelnen Ländern. In Deutschland verwaltet das DENIC (Deutsches Network Information Center, `http://www.DENIC.de`) die IP-Adressen.

Private IP-Adressen

Einige IP-Adressen sind von der globalen Zuweisung ausgenommen. Sie werden im Internet ignoriert und stehen für private Netzwerke zur Verfügung. Unter anderem sind das die folgenden Netzwerke:

- 10.0.0.0/8
- 127.0.0.0/8
- 172.16.0.0/12
- 169.254.0.0/16
- 192.168.0.0/16

Die Ausgabe des Programms `IPAddresses` (Listing 7.3) zeigt, dass die Schnittstelle `eth0` direkt mit dem Internet verbunden ist, `lo`, `tap0` und `wlan0` dagegen nicht. `127.0.0.1` liegt im privaten Netzwerk `127.0.0.0/8`, `10.4.0.1` in `10.0.0.0/8` und `192.168.1.23` in `192.168.0.0/16`

---

7 Die Zeile zur Schnittstelle `lo` weist in diesem Beispiel 0 Bit für die Netzwerkadresse aus. Diese Angabe hat keine Bedeutung, weil `lo` keine „echte“ Netzwerkschnittstelle ist, wie unten erklärt wird.

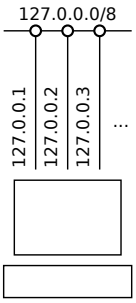
8 Genauer gesagt ist die IANA (*Internet Assigned Numbers Authority*), die der ICANN untersteht, für IP-Adressen und Domainnamen zuständig.

7.1.5 Loopback-Device

Pseudo-Netzwerkschnittstelle zum gefahrlosen Testen

Um mit Netzwerksoftware zu arbeiten, sind eigentlich wenigstens zwei Rechner notwendig. Zur Vereinfachung gibt es auf allen Systemen ein sogenanntes **Loopback-Device**, eine virtuelle Netzwerkschnittstelle. Dieses „Gerät“ existiert nicht in Hardware. Es simuliert einen Netzwerkadapter, der in einer Schleife mit dem *eigenen System* verbunden ist. Das Loopback-Device ist unter Namen wie `lo`, `lo0` oder `loopback` erreichbar.

Das Loopback-Device ist an das Netzwerk `127.0.0.0/8` „angeschlossen“. Dieser Schnittstelle sind automatisch *alle* Hostadressen des Netzwerks zugeordnet.[9] Üblicherweise verwendet man zwar die IP-Adresse `127.0.0.1`, aber beispielsweise ist auch `127.44.55.66` eine IP-Adresse des eigenen Systems.



Erkennen des Loopback-Device

Der Getter `isLoopback` der Klasse `NetworkInterface` erkennt ein Loopback-Device unabhängig vom Namen. Das folgende Programm gibt die Daten des Loopback-Device aus, wie immer es heißen mag:

```
import java.net.*;
import java.util.*;

public class FindLoopback {
    public static void main(final String... args) throws SocketException {
        for(final Enumeration<NetworkInterface> interfaces =
                    NetworkInterface.getNetworkInterfaces();
                interfaces.hasMoreElements();) {
            final NetworkInterface networkInterface = interfaces.nextElement();
            if(networkInterface.isLoopback())
                for(InterfaceAddress address: networkInterface.getInterfaceAddresses())
                    System.out.printf("%s\t%s%n",
                                      networkInterface.getDisplayName(),
                                      address.getAddress());
        }
```

[9] Das ist ein Beispiel einer einzelnen Schnittstelle mit mehreren, in diesem Fall sogar sehr vielen Hostadressen.

```
    }
}
```

Listing 7.4: Loopback-Netzwerkschnittstelle und IP-Adressen ausgeben.

Das Loopback-Device eignet sich zum Entwickeln und Testen von Netzwerksoftware ohne echtes Netzwerk.

7.1.6 Symbolische Hostnamen

Eingängiger Ersatz für IP-Adressen

IP-Adressen reichen im Grunde aus, um einen Rechner im Internet zu identifizieren. Allerdings lassen sich vierteilige Codes nicht gerade leicht merken. Daher haben sich **symbolische Hostnamen** durchgesetzt, die nichts anderes als eingängige Synonyme für numerische IP-Adressen sind.

Symbolische Hostnamen[10] sind Strings aus Elementen, die mit Punkten getrennt sind, wie zum Beispiel:[11]

```
sol.cs.hm.edu
```

Hierarchische Organisation in Domains

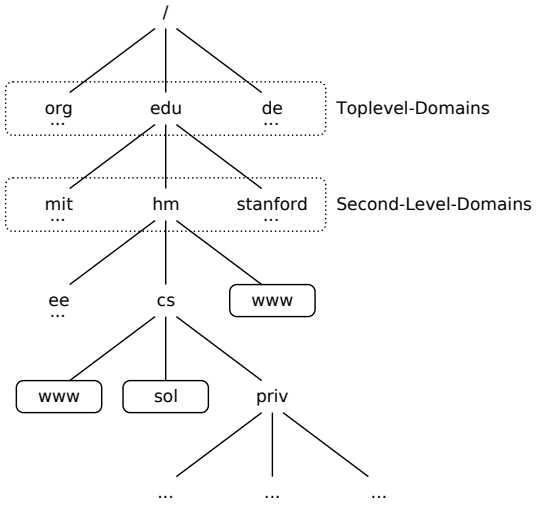
Hostnamen sind in einem Baum organisiert. Wohlgemerkt, es geht hier nur um die Organisation der Host*namen*, nicht um die Rechner mit diesen Namen.

Von hinten nach vorne gelesen definieren die Elemente eines Hostnamens einen Pfad, der bei der Wurzel des Namensraums beginnt und an einem Blatt endet.[12] Der Hostname `sol.cs.hm.edu` führt von der anonymen Wurzel zuerst zum Knoten `edu`, der auf der obersten Ebene (*top level*) liegt, von dort zum Knoten `hm`, weiter hinab zum Knoten `cs` und schließlich zum Knoten `sol`:

[10] Der Begriff „Hostname“ steht im Weiteren immer für „symbolische Hostnamen“, der Begriff „IP-Adresse“ für eine „numerische IP-Adresse“.

[11] Die Punkte in Hostnamen haben nichts mit den Punkten in IP-Adressen zu tun! Der Rechner mit dem Hostnamen `sol.cs.hm.edu` hat die IP-Adresse `129.187.208.11`. Aber dabei entspricht `sol` *nicht* dem ersten Byte 129, `cs` dem zweiten Byte 187 und so weiter.

[12] Die Schreibweise ist also gerade andersherum als bei Pfadnamen im Filesystem. Dort steht das oberste Directory vorne und der Filename selbst hinten.



Als **Domainname** wird jeder Pfad in diesem Baum bezeichnet, ob er bei einem Blatt endet oder nicht. Im letzten Fall steht er für einen ganzen Teilbaum. Beispielsweise ist `hm.edu` der Domainname der Hochschule München. Ein „Hostname“ ist ein Domainname, der an einem Blatt endet und damit genau einen Host identifiziert.

Toplevel- und untergeordnete Domains

Domainnamen sind weltweit abgestimmt, ebenso wie IP-Adressen, und stehen unter der Obhut der ICANN. Besonders defensiv verfährt die ICANN mit den Namenselementen auf der obersten Ebene, den **Toplevel-Domains** [13] (TLD). Die meisten Toplevel-Domains sind Länderkürzel[14], wie zum Beispiel `de` für Deutschland oder `li` für Liechtenstein. Darüber hinaus gibt es eine Reihe von „generischen Toplevel-Domains“ für Organisationsformen und Geschäftsbereiche. Beispiele sind `edu` für Bildungseinrichtungen, `org` für nichtkommerzielle Organisationen und `post` für Logistikunternehmen. Darüber hinaus sind einige TLDs für besondere Zwecke reserviert, wie `test` zum Testen und `invalid` als ausdrücklich ungültige Domain.[15]

Wie bei IP-Adressen kümmern sich nationale Organisationen um die Domainnamen unter der entsprechenden Toplevel-Domain. So verwaltet das DENIC außer IP-Adressen auch die Domainnamen in der Toplevel-Domain `de`. Diese Delegation setzt sich Ebene für Ebene fort. Die Domainnamen unter `hm.edu` werden innerhalb der Hochschule München vergeben. Beispielsweise ist `cs.hm.edu` der Domainname der Fakultät für Informatik und Mathematik an der Hochschule München. Die Namen in der Domain `cs.hm.edu` organisiert wiederum die Fakultät in eigener Regie.

[13] Die Liste aller TLDs ist auf `http://www.iana.org/domains/root/db` zu finden.

[14] Ländernamen und TLDs listet `http://www.iso.org/iso/list-en1-semic-3.txt` auf.

[15] Erschöpfende Auskunft gibt `http://tools.ietf.org/html/rfc2606`.

Die Schachtelungstiefe von Domains ist offen. Allerdings sind Domainnamen auf maximal 253 Zeichen Länge begrenzt.

7.1.7 Domainname-Server (DNS)

Verknüpfung von Domainnamen mit IP-Adressen

Die Protokolle von Ebene 1 bis 3 kennen nur IP-Adressen und wissen nichts von Hostnamen. Hostnamen müssen daher vor dem Transport in IP-Adressen „übersetzt“ werden, sonst finden sie ihren Bestimmungsort nicht. Für diese Übersetzung sind bestimmte Rechner im Netzwerk zuständig, die **DNS-Server** (*Domain Name System Server*).

Mit Kenntnis eines Hostnamens wendet sich ein Absender also zuerst an einen DNS-Server, der die IP-Adresse des Hostnamens herausfindet.[16] Mit der IP-Adresse nimmt der Rechner dann Kontakt zum Zielsystem auf. Es gibt viele DNS-Server, die selbst hierarchisch organisiert sind. Wenn ein DNS-Server die IP-Adresse eines Hostnamens nicht kennt, gibt er die Frage an seinen übergeordneten DNS-Server weiter. Die letzte Instanz bilden sogenannte **Root-Nameserver**. Diese Root-Nameserver sind kritisch für das Funktionieren des gesamten Internets.[17]

Die Namensauflösung über DNS-Server geschieht in der Regel unbemerkt auf der Ebene des Betriebssystems. Sie entfällt, wenn die IP-Adresse des Zielrechners bereits bekannt ist.[18] Zum Hostnamen `localhost` gehört beispielsweise immer die IP-Adresse `127.0.0.1`. Dazu wird kein DNS-Server befragt.

7.1.8 Ports

Aufteilung des Netzwerkverkehrs eines Hosts

Auf einem vernetzten Rechner sind oft mehrere Programme aktiv, die alle gleichzeitig über das Netzwerk mit verschiedenen anderen Rechnern kommunizieren. Die Daten laufen dabei alle über dieselbe Netzwerkschnittstelle. Um den Datenverkehr zu ordnen, werden Pakete auf Ebene 4 des Protokollstapels[19] auf verschiedene **Ports** aufgeteilt. Ports sind vorzeichenlose 16-Bit-Zahlen, also Werte im Bereich 0 bis 65535.

[16] Jeder Rechner muss zumindest *eine* numerische IP-Adresse kennen, nämlich die „seines“ DNS-Servers. Meist halten Rechner sicherheitshalber zwei oder drei IP-Adressen von DNS-Servern vor.

[17] Heute sind diese Server physisch über die ganze Welt verteilt. Die Kontrolle über einen Root-Nameserver hat politische Aspekte.

[18] Die lokale Datei `hosts` enthält eine statische Abbildung von Hostnamen auf IP-Adressen. Sie eignet sich für genau bekannte Kommunikationspartner und erspart den Weg über DNS-Server. Wo die `hosts`-Datei zu finden ist, hängt vom System ab. Eine Suche im Filesystem fördert die Datei zutage.

[19] Genau genommen sind Ports ein Merkmal der Protokolle TCP und UDP auf Ebene 4. Andere Protokolle müssen nicht damit arbeiten.

Jedes Programm, das Daten vom Netzwerk empfangen will, legt dafür einen eindeutigen Port fest.[20] Das Betriebssystem sortiert alle einlaufenden Pakete[21] nach Portnummern und stellt an jedes Programm genau die Pakete zu, die an dessen Port adressiert sind. Pakete an fremde Ports bekommt ein Programm nicht zu Gesicht.

Der Sender muss neben dem Zielrechner auch den Port auf dem Zielrechner kennen, um das richtige Empfängerprogramm zu erreichen. *Wie* sich Sender und Empfänger auf eine Portnummer einigen, regelt das Protokoll nicht. Ohne eine passende Portnummer kommt keine Verbindung zustande.

Vereinbarte Portnummern für bestimmte Dienste

Einige Ports sind für bestimmte Zwecke[22] festgelegt, um Abreden einzusparen. Der Katalog dieser **well-known Ports** ist von der IANA standardisiert.[23] Hier einige Beispiele:

Portnummer	Dienst	Zweck
7	Echo	Schickt die empfangenen Daten unverändert wieder zurück
13	Time	Antwortet mit der aktuellen Zeit
21	File Transfer	Übertragung von Dateien
22	Secure Shell	Abgesichertes Login
25	E-Mail Transmission	Transport von E-Mail
53	Domain Name Service	Liefert IP-Adressen zu Hostnamen
80	Webservice	Liefert Webseiten
443	Secure Webservice	Liefert Webseiten verschlüsselt

Portnummern für vertrauenswürdige Dienste

Unabhängig von wohlbekannten und weniger bekannten Ports genießen die Ports mit Nummern bis 1024 in den meisten Betriebssystemen eine Sonderbehandlung.[24] Nur Programme mit Administratorrechten dürfen Daten auf diesen **priviledged Ports** entgegennehmen. Die meisten *well-known* Ports liegen in diesem Bereich. Das

[20] In gewissen Grenzen darf sich ein Programm selbst einen Port aussuchen. In erster Linie muss der Port allerdings „frei“ sein, das heißt, er darf von keinem anderen Programm benutzt werden.

[21] Auch für ausgehende Verbindungen werden Ports benutzt. Diese Portnummern spielen allerdings in der Regel weiter keine Rolle und werden vom Betriebssystem kurzfristig automatisch zugewiesen. Sie liegen in der Regel im Bereich ab 49152 = 0xC000.

[22] Technisch sind alle Ports gleichwertig. Sofern sich alle Beteiligten darüber einig sind, kann jeder Port für jeden Zweck benutzt werden. Eine Kontrolle gibt es nicht.

[23] Die Liste dieser abgestimmten Portnummern findet sich auf `http://www.iana.org/assignments/port-numbers`.

[24] Anbieter von Diensten können versuchen, Portnummern zwischen 1024 und 49151 bei der IANA für ihre Zwecke als *well known* registrieren zu lassen. Die verbleibenden Portnummern ab 49152 (= 0xC000) bis 65535 (= 0xFFFF) können nicht registriert werden und stehen für allgemeine Zwecke zur Verfügung. Beispielsweise erhalten ausgehende Verbindungen automatisch gerade freie Portnummern aus diesem Bereich. Sie werden als *ephemeral* Ports bezeichnet.

ist eine Schutzmaßnahme gegenüber Besuchern, die auf eine gewisse Integrität der Dienste auf diesen Ports vertrauen können.[25]

Ports sind eine logische Konstruktion, um den gesamten Datenverkehr eines Rechners besser zu organisieren. Sie haben keine physische Entsprechung. Auf Ebene 3 des Schichtenmodells und darunter gibt es keine Ports.

7.1.9 Dienste und Protokolle

Protokolle regeln Ablauf der Kommunikation

Auf den Ports eines Rechners bieten verschiedene Programme ihre Dienste an, die auch als **Services** bezeichnet werden. Ein und derselbe Rechner kann ganz verschiedene Services bereitstellen, die nichts miteinander zu tun haben und nichts voneinander wissen. Der Begriff „Server“ wird manchmal für ein einzelnes Programm gebraucht und manchmal auch für den ganzen Rechner, auf dem solche Programme laufen.

Um einen Dienst in Anspruch zu nehmen, müssen sich die beteiligten Systeme über den Ablauf der Kommunikation einig sein. Ein **Protokoll** regelt bis ins Detail

- die zeitliche Abfolge und Flussrichtung der Nachrichten,
- die an jedem Punkt der Kommunikation zulässigen Nachrichten und
- die Darstellung der Nachrichten in Form von Bits und Bytes.

Die Protokolle der *well-known* Ports sind standardisiert, sonst wäre eine reibungslose Kommunikation nicht möglich. Die folgende Liste zeigt einige Protokolle:

Portnummer	Dienst	Protokoll
7	Echo	-
13	Time	-
21	File Transfer	FTP (*File Transfer Protocol*)
22	Secure Shell	SSH
25	E-Mail Transmission	SMTP (*Simple Mail Transfer Protocol*)
53	Domain Name Service	DNS
80	Webservice	HTTP (*Hypertext Transfer Protocol*)
443	Secure Webservice	HTTPS (*Hypertext Transfer Protocol Secure*)

Jedem Service und damit jedem Port ist ein bestimmtes Protokoll zugeordnet.

[25] Welches Programm auch immer auf einem *well-known* Port antwortet, es erfordert zumindest Administratorrechte auf dem betreffenden System. Ein beliebiger Benutzer ohne besondere Rechte kann dort keine Dienste anbieten.

Protokolle können sehr einfach sein, wie zum Beispiel das Protokoll des Echo-Service: Alle Bytes, die der Server empfängt, schickt er in der gleichen Reihenfolge wieder zurück. Aber selbst dieses scheinbar triviale Protokoll trifft eine Regelung: Es legt fest, dass zuerst der Anrufer Daten senden muss und erst dann eine Antwort zurückkommt. Sollte der Client dieses Protokoll nicht einhalten und beispielsweise nach dem Verbindungsaufbau keine Daten senden, sondern schweigen, dann kommt keine Kommunikation zustande.

Dem stehen komplexere Protokolle gegenüber, wie zum Beispiel FTP, deren Beschreibung längere Dokumente einnimmt und deren vollständige Implementierung einigen Aufwand erfordert.

7.1.10 Client-Server-Systeme

Rollenaufteilung bei Netzwerkkommunikation

Um in einem Netzwerk sinnvoll zusammenzuarbeiten, müssen sich die beteiligten Rechner organisieren. Zwei verbreitete Organisationsformen sind Client-Server-Systeme und Peer-to-Peer-Netzwerke. Sie verteilen die Rollen unter den Kommunikationspartnern auf unterschiedliche Art.

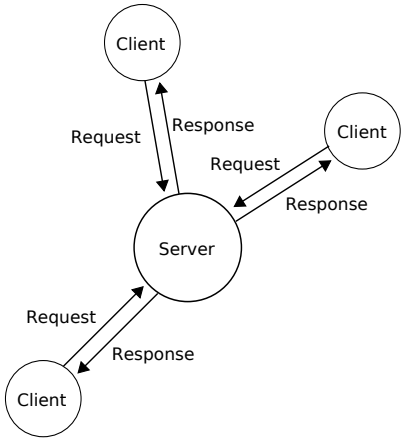
In **Client-Server-Systemen** sind die Rollen asymmetrisch zugewiesen. Einer der Rechner im Netzwerk ist der Server, der Dienste bereitstellt oder über zentrale Daten verfügt. Alle anderen Rechner sind Clients, die die Dienste des Servers in Anspruch nehmen und seine Daten abrufen.

Aktive Clients und passive Server

Die Clients sind dabei die aktiven Teilnehmer, die Anfragen (*request*) an den Server schicken. Der Server ist passiv und tut von sich aus nichts. Erst wenn Clients Anfragen schicken, sendet ein Server Antworten (*response*) zurück. Ein Server ist nicht auf Clients angewiesen. Ob es überhaupt Clients gibt und wie viele gegebenenfalls existieren, wann sie starten und wieder enden, spielt für den Server keine Rolle. Die Clients sind dagegen ohne den Server hilflos.

In einem solchen System kennt jeder Client „seinen“ Server. Dagegen kennen sich die Clients untereinander in der Regel nicht, ebenso wenig wie der Server die Clients kennt.[26]

[26] Das gilt nur für Server, die *jeden* Client bedienen. Viele Server erbringen ihre Leistung nur gegenüber Clients, die ihre Berechtigung in irgendeiner Weise nachweisen können. In diesem Szenario kann der Server verschiedene Clients sehr wohl unterscheiden.



Einfache Verwaltung, Server ist kritisch

Die zentralisierte Struktur macht Client-Server-Systeme einfach zu verwalten. Die Clients verfügen in der Regel über keine Daten[27] und Funktion. Das „Wohlbefinden“ eines einzelnen Clients spielt für die Struktur als Ganzes weiter keine Rolle. Nur der zentrale Server muss sorgfältig gehegt und gepflegt werden.

Darin liegt auch der größte Nachteil dieser Systeme: Wenn der Server ausfällt, sind alle Clients zur Untätigkeit verurteilt. Im Grunde ist das System damit ausgefallen.

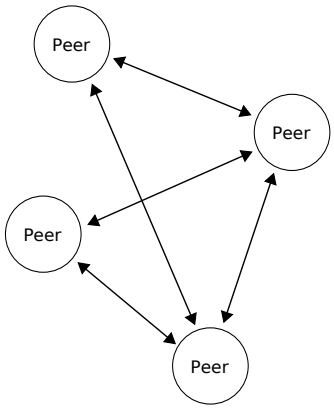
Die meisten Netzwerkanwendungen sind so organisiert, wie zum Beispiel die E-Mail-Kommunikation: Clients sind die Mail-Reader, die ihre Nachrichten über Mailserver austauschen. Sogar das ganze WWW arbeitet grundsätzlich als Client-Server-System: Clients sind die Webbrowser, Server eben die Webserver.

7.1.11 Peer-to-Peer-Netzwerke

Gleichrangige Teilnehmer

In einem **Peer-to-Peer-Netzwerk** sind alle Rechner gleichberechtigt. Jeder Rechner kann mit jedem anderen Kontakt aufnehmen und Daten austauschen. Jeder Beteiligte ist gewissermaßen Client und Server zugleich.

[27] Oft puffern Clients Daten kurzfristig, um den Netzwerkverkehr zu begrenzen. Diese Caches sind aber redundant und müssen nicht gesichert werden.



Neue Rechner, die an einem Peer-to-Peer-Netzwerk teilnehmen möchten, wenden sich an einen bereits bekannten Teilnehmer oder schicken eine pauschale Suchanfrage in das Netzwerk.

Typische Anwendungen von Peer-to-Peer-Netzwerken sind Chat-Systeme und Datei-Austausch-Dienste. In Peer-to-Peer-Netzwerken gibt es in der Regel keinen einzelnen Teilnehmer, der den kompletten Datenbestand speichert oder auch bloß kennt. Informationen werden nach Bedarf verteilt und transportiert.

Dezentrale Datenhaltung, robust gegen Ausfälle

Peer-to-Peer-Netzwerke sind robuster als Client-Server-Systeme, weil der Ausfall einzelner Systeme den Rest des Systems nicht unbedingt beeinträchtigt. Im Extremfall reicht es aus, wenn alle bis auf zwei beliebige Teilnehmer übrig bleiben, die immer noch miteinander arbeiten können.

Anforderungen legen Architektur fest

Client-Server-Systeme und Peer-to-Peer-Netzwerke sind keine austauschbaren Alternativen. In der Regel diktiert der Anwendungszweck eine der Architekturen und schließt die andere aus. Schließlich gibt es Mischformen, wie zum Beispiel Server, die Kontakte herstellen und sich dann ausklinken. Die weitere Kommunikation läuft direkt zwischen Hosts und ohne Server ab.

7.2 Clients und Server

7.2.1 Sockets

`Socket` repräsentiert Endpunkt einer Netzwerkverbindung

Die Klasse `Socket` im Package `java.net` ist der Grundbaustein für die Netzwerkkommunikation via TCP.[28] Die wörtliche Übersetzung, „Steckdose“, illustriert gut

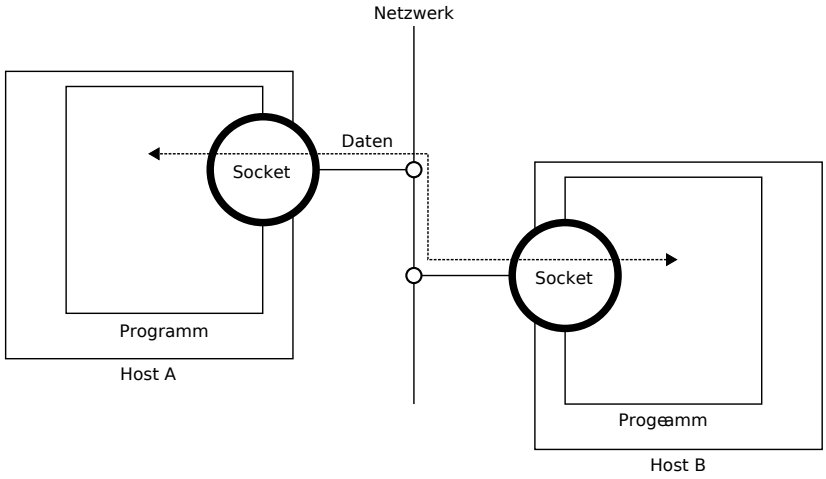
[28] Das Transportprotokoll UDP baut auf der Klasse `DatagramSocket` auf, das Transportprotokoll SCTP verwendet wiederum andere Klassen.

die Idee: Aus der Sicht eines Programms spielt ein `Socket`-Objekt die Rolle einer Steckdose in der Wand, die auf geheimnisvolle Art mit einem unsichtbaren, dahinterliegenden Leitungssystem verknüpft ist.

Zwei Programme, die miteinander über ein Netzwerk Daten austauschen, benutzen jedes einen Socket. Was immer ein Programm in den eigenen „lokalen“ Socket schickt, kommt aus dem korrespondierenden entfernten Socket wieder zum Vorschein.

Sockets kennen keine Richtung und kein Protokoll

Sockets arbeiten in beiden Richtungen, sie sind „bi-direktional“. Jedes Programm kann Informationen in den Socket schreiben und aus dem Socket lesen. Die Abfolge der Schreib- und Leseoperationen ist Sache der beiden Programme, die sich darüber einig sein müssen. Sockets transportieren nur Bytes. Sie kennen kein Protokoll und keine Richtung.



Sockets nur paarweise sinnvoll

Sockets sind immer paarweise verbunden. Ein einzelner Socket ist nutzlos, ebenso wie drei oder mehr Sockets nicht verbunden werden können.[29] Ein Programm spricht über seinen eigenen Socket mit *einem* anderen Programm über dessen Socket.

Konstruktor mit Hostnamen und Portnummer

Die hier beschriebenen Sockets werden auch als **Client-Sockets** bezeichnet.[30] Der `Socket`-Konstruktor erwartet die IP-Adresse des anderen Rechners und die Portnummer auf diesem Rechner, wie zum Beispiel:

```
Socket socket = new Socket("129.187.208.11", 80);
```

[29] Davon gibt es Ausnahmen. Beim *Broadcasting* schickt ein Rechner Daten, die mehrere andere gleichzeitig empfangen. Das sind allerdings Spezialanwendungen, die hier nicht zur Diskussion stehen.

[30] Es gibt weitere Arten von Sockets für andere Zwecke als Datenaustausch. In Abschnitt 7.2.8 kommen beispielsweise „Serversockets“ ins Spiel, die zum Verbindungsaufbau gebraucht werden.

Alternativ akzeptiert der Konstruktor einen symbolischen Hostnamen.[31] In beiden Fällen ist das erste Argument ein String.[32]

```
Socket socket = new Socket("sol.cs.hm.edu", 80);
```

Fehlerquellen beim Aufbau der Verbindung

Die Verbindung wird im Konstruktor aufgebaut. Er scheitert mit einer Exception, wenn der Kontakt nicht gelingt, wie zum Beispiel:

- `IOException` – die Verbindung zum Netzwerk konnte nicht hergestellt werden,
- `UnknownHostException` – das Netzwerk funktioniert, aber die IP-Adresse oder der Hostname sind nicht bekannt,
- `ConnectException` – der Host existiert, aber er verweigert die Annahme der Verbindung zum gewünschten Port.

Sockets müssen mit einem Aufruf von `close` geschlossen werden, ebenso wie I/O-Streams.

Test der Netzwerkverbindung

Das folgende Beispielprogramm öffnet eine Verbindung zu einem Rechner und schließt sie sofort wieder. Hostname und Port werden auf der Kommandozeile angegeben. Das Programm schickt keine Daten und versucht auch nicht, Daten zu empfangen.

```
import java.io.*;
import java.net.*;

public class ConnectPort {
    public static void main(String... args) throws IOException {
        String hostname = args[0];
        int port = Integer.parseInt(args[1]);
        try(Socket socket = new Socket(hostname, port)) {
            System.out.println("connected");
        }
    }
}
```

Listing 7.5: Socket öffnen zu Server und Port laut Kommandozeile.

Mit dem Programm kann überprüft werden, ob ein Rechner grundsätzlich antwortet:

---

[31] Unsichtbar für den Anwender wird der Hostname an einen DNS-Server geschickt (siehe Seite 459) und in eine IP-Adresse aufgelöst. Der Rest läuft ab wie beim ersten Konstruktor.

[32] Eine IP-Adresse wird ebenso wie ein Hostname als String angegeben, obwohl es sich logisch um einen 32-Bit-Wert handelt, der genau einem Wert vom primitiven Typ `int` entspricht.

```
$ java ConnectPort www.google.com 80
connected
$ java ConnectPort 129.187.208.11 80
connected
$ java ConnectPort sol.cs.hm.edu 80
connected
$ java ConnectPort sim.sala.bim.invalid 80
Exception in thread "main" java.net.UnknownHostException: sim.sala.bim.invalid
$ java ConnectPort www.google.com 81
Exception in thread "main" java.net.ConnectException: Connection timed out
```

Beim letzten Aufruf des Programms dauert es eine Weile, bis die Fehlermeldung erscheint. Regulär sollte ein Server eine Verbindung auf einem bestimmten Port akzeptieren oder verweigern und diese Entscheidung dem Client mitteilen. Der `Socket`-Konstruktor liefert dann entweder ein `Socket`-Objekt oder wirft eine Exception. In letzten Beispiel schweigt der Server allerdings, das heißt, er schickt weder eine positive noch eine negative Antwort, sondern überhaupt keine.[33] Der normale `Socket`-Konstruktor blockiert in diesem Fall für unbestimmte Zeit, die vom Betriebssystem und anderen Einflussfaktoren abhängt. Dieses Verhalten ist nicht immer brauchbar.

Kontaktversuch mit maximaler Wartezeit

Die Wartezeit bis zum Abbruch eines Verbindungsversuchs lässt sich explizit kontrollieren. Dazu muss die Arbeit des regulären `Socket`-Konstruktors allerdings in einzelne Schritte zerlegt werden, die sich dann genauer steuern lassen.

1. Zuerst wird ein *unverbundenes* `Socket`-Objekt erzeugt, das nur als Objekt in der JVM lebt und noch nicht mit der Außenwelt in Verbindung steht.

   ```
   Socket socket = new Socket();
   ```

2. Ein Objekt der Klasse `InetSocketAddress` kapselt einen Hostnamen und eine Portnummer. Auch das ist nur ein Objekt auf dem Heap, das keinen Kontakt zum Netzwerk hat:

   ```
   InetSocketAddress isa = new InetSocketAddress(hostname, port);
   ```

3. Ein Aufruf der `Socket`-Methode `connect` versucht schließlich, die eigentliche Verbindung herzustellen. Diese Methode akzeptiert eine maximale Zeitspanne in Millisekunden, nach der der Zielrechner spätestens reagiert haben muss:

   ```
   socket.connect(isa, millis);
   ```

Das folgende Programm erfüllt den gleichen Zweck wie `ConnectPort` (Listing 7.5), erwartet aber als drittes Kommandozeilenargument eine maximale Wartezeit in Millisekunden:

[33] Dieses Verhalten wird oft mit Vorsatz von einem Schutzmechanismus, wie zum Beispiel einer Firewall, herbeigeführt, um Portscanning zu behindern.

```
import java.io.*;
import java.net.*;

public class ConnectPortTimed {
    public static void main(String... args) throws IOException {
        String hostname = args[0];
        int port = Integer.parseInt(args[1]);
        int timeoutMillis = Integer.parseInt(args[2]);
        try(Socket socket = new Socket()) {
            InetSocketAddress address = new InetSocketAddress(hostname, port);
            socket.connect(address, timeoutMillis);
            System.out.println("connected");
        }
    }
}
```

Listing 7.6: Socket mit Timeout öffnen zu Server und Port.

Dieses Programm liefert zuverlässig innerhalb der vorgegebenen Zeitspanne ein Ergebnis (ob positiv oder nicht). Es bleibt aber keinesfalls für unbestimmte Zeit „hängen“:

```
$ java ConnectPortTimed www.google.com 80 100
connected
$ java ConnectPortTimed www.google.com 81 100
Exception in thread "main" java.net.SocketTimeoutException: connect timed out
```

7.2.2 Portscanner

Absuchen der Ports eines Hosts

Die meisten Server bieten nur die erforderlichen Dienste an, erbringen darüber hinaus aber keine freiwilligen Zusatzleistungen. Diese Sparsamkeit dient der Sicherheit, weil jeder verfügbare Dienst missbraucht werden könnte. Manchmal reicht es schon aus, einen weniger leistungsschwachen Server von vielen verschiedenen Clients aus gleichzeitig mit sinnlosen Anfragen zu bombardieren. Lastet man den Server damit so weit aus, dass dieser nur noch langsam oder zeitweise überhaupt nicht mehr reagiert, so spricht man von einem DDoS-Angriff (*distributed denial of service attack*).

Das folgende Programm öffnet auf allen Ports Verbindungen zu einem Server und protokolliert die erfolgreichen Versuche. Manche Server beantworten Kontaktversuche auf gesperrten Ports überhaupt nicht. Der `Socket`-Konstruktor blockiert in diesem Fall für eine gewisse Zeit. Um das Scannen weiterer Ports nicht aufzuhalten, startet das Programm für jeden Port einen Thread (Kapitel 6). Allerdings kommt nicht jede JVM mit 2^{16} = 65536 Threads zurecht. Deshalb begrenzt das Programm

die Anzahl der gleichzeitig aktiven Threads.[34]

```
import java.io.*;
import java.net.*;

public class PortScan extends Thread {
    private static String hostname;

    private final int port;

    private static int numThreads = 0;

    private static final int MAX_THREADS = 100;

    public PortScan(int port) throws InterruptedException {
        synchronized(hostname) {
            while(numThreads >= MAX_THREADS)
                hostname.wait();
            numThreads++;
        }
        this.port = port;
    }

    public void run() {
        try(Socket socketNotUsed = new Socket(hostname, port)) {
            System.out.println(port);
        }
        catch(IOException iox) {
            // explizit ignoriert
        }
        synchronized(hostname) {
            numThreads--;
            hostname.notify();
        }
    }

    public static void main(String... args) throws InterruptedException {
        hostname = args[0];
        for(int port = 0; port < 65536; port++)
            new PortScan(port).start();
    }
}
```

Listing 7.7: Verbindungen zu allen Ports öffnen und die erfolgreichen Versuche protokollieren.

Ein Programmstart auf dem System des Autors ergibt bei einer Laufzeit von einigen Sekunden folgendes Ergebnis:[35]

[34] Die Laufzeitbibliothek bietet für solche Zwecke sogenannte Thread-Pools an. Hier kommt man aber auch mit einfacheren Mitteln zum Ziel.

[35] Der Rechner steht in diesem Beispiel in einem geschützten privaten Netzwerk und kann Dienste ohne Gefahr anbieten. Aus dem Internet gesehen sollte die Liste wesentlich kürzer ausfallen.

```
$ java PortScan localhost
37
13
9
7
21
23
111
22
631
1059
2049
3690
6000
8118
9050
37649
43592
50379
58205
```

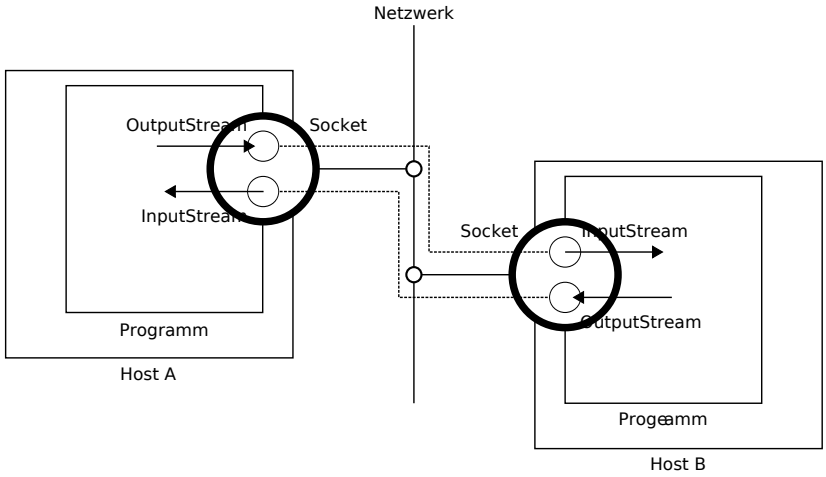
Möglicherweise Auslegung als Angriffsversuch

Lassen Sie dieses Programm nicht auf beliebige Rechner im Internet los! Solche systematischen Portabfragen können als Angriffsversuch ausgelegt werden und Rückfragen nach sich ziehen.

7.2.3 Lesen und Schreiben

I/O-Streams zum Datentransport

Sockets repräsentieren die Endpunkte einer Verbindung, transportieren aber selbst keine Nutzdaten. Der eigentliche Datenaustausch verläuft über ein Paar von zwei gegenläufigen I/O-Streams (siehe Kapitel 1.2).



Diese Streams, ein `InputStream` zum Lesen und ein `OutputStream` zum Schreiben, liefern die beiden folgenden `Socket`-Getter:[36]

```
InputStream  getInputStream()


OutputStream  getOutputStream()
```

Der Umgang mit diesen Streams unterscheidet sich nicht von anderen I/O-Streams. Das folgende minimale Client-Programm zeigt das Prinzip:

Rumpf eines Netzwerk-Clients

```
import java.io.*;
import java.net.*;

public class NullClient {
    public static void main(String... args) throws IOException {
        String hostname = args[0];
        int port = Integer.parseInt(args[1]);
        try(Socket socket = new Socket(hostname, port);
            InputStream input = socket.getInputStream();
            OutputStream output = socket.getOutputStream()) {
            // Protokoll abwickeln
        }
    }
}
```

Listing 7.8: Rumpf eines Client-Programmes.

Das folgende Programm liest Text von einem Server und gibt ihn auf dem Bildschirm aus:

```
import java.io.*;
import java.net.*;

public class PrintTextResponse {
    public static void main(String... args) throws IOException {
        String hostname = args[0];
        int port = Integer.parseInt(args[1]);
        try(Socket socket = new Socket(hostname, port);
            InputStream input = socket.getInputStream();
            InputStreamReader reader = new InputStreamReader(input);
            BufferedReader breader = new BufferedReader(reader)) {
            for(String line = breader.readLine(); line != null; line = breader.readLine()
                System.out.println(line);
        }
```

[36] Der konkrete Typ der beiden Objekte bleibt im Dunkeln. Sie sind jedenfalls kompatibel zu den betreffenden Basisklassen.

```
    }
}
```

Listing 7.9: Textantwort von einem Server lesen und ausgeben.

Kontakt zu einem öffentlichen Zeit-Service

Manche Maschinen bieten auf *well-known* Port 13 (siehe Seite 461) den Daytime-Service an. Das entsprechende Protokoll ist einfach: Der Server schickt sofort eine Antwort mit der aktuellen Zeitangabe in Textform. Er erwartet keine Anfrage. Das Programm `PrintTextResponse` (Listing 7.9) implementiert genau dieses Protokoll.

Ein Rechner, der den Daytime-Service anbietet, ist beispielsweise `time.nist.gov`, ein Server des „National Institute of Standards and Technology“ (NIST) in Colorado, USA.[37]

```
$ java PrintTextResponse time.nist.gov 13
55883 11-11-18 13:57:35 00 0 0  58.8 UTC(NIST) *
```

Die Angabe in der Antwort (11-11-18, 13:57:35) bezieht sich auf die Weltzeit (*Uniform Time Coordinated*, UTC).[38] Für eine lokale Zeitangabe ist noch die Zeitzone zu berücksichtigen.

7.2.4 Beenden einer Verbindung

Schließen von Streams und Sockets sehr wichtig

An I/O-Streams sind in der Regel externe Ressourcen gebunden, die außerhalb der Kontrolle der JVM liegen. Aus diesem Grund *muss* die `Stream`-Methode `close` aufgerufen werden (Seite 34). Für `Sockets` gilt das ebenso. Vielleicht ist das korrekte Schließen von Sockets sogar noch wichtiger, weil nicht nur Ressourcen des lokalen Systems betroffen sind, sondern auch noch Ressourcen anderer Systeme draußen im Netzwerk.

Ebenso wie Streams implementiert die Klasse `Socket` das Interface `Closeable` (Seite 38) und eignet sich damit für das ARM (Seite 36). Üblicherweise werden Sockets, gleichrangig neben Streams und anderen I/O-Objekten, in ARM-Köpfen definiert. Die Programme `ConnectPort` (Listing 7.5) und `PrintTextResponse` (Listing 7.9) setzen das beispielsweise um.

Schließen eines Streams schließt auch Socket

Ein Socket verständigt sich direkt mit seinen beiden zugeordneten Stream-Objekten. Schließt man ein beliebiges der drei Objekte, dann schließt man automatisch auch

[37] `time.nist.gov` ist nicht immer erreichbar. Eine Liste von weiteren Servern finden Sie auf `http://tf.nist.gov/tf-cgi/servers.cgi`.

[38] Die Auslegung des Akronyms UTC ist nicht ganz einheitlich.

die anderen beiden. Für mögliche Dekoratoren der Streams gilt das nicht, sie müssen explizit geschlossen werden.[39]

Das Interface `Closeable` erfordert eine idempotente `close`-Methode.[40] Deshalb ist es in Ordnung, wenn alle Beteiligten, also ein Socket und auch seine Streams, nacheinander und alle einzeln geschlossen werden. Die Reihenfolge der `close`-Aufrufe spielt wegen der Idempotenz keine Rolle.

Fehler bei einseitig geschlossener Verbindung

Netzwerkverbindungen haben immer zwei Enden mit einem `OutputStream` beim Sender und einem `InputStream` beim Empfänger. Dabei gibt es unterschiedliche Fehlersituationen:

- Lesen und Schreiben von einem Socket-Stream, der lokal geschlossen wurde, wirft eine `SocketException` („Socket closed“).
- Lesen von einem Socket-Stream, dessen fernes Ende geschlossen ist, liefert den Fluchtwert -1 (siehe Seite 30).
- Schreiben auf einen Socket-Stream, dessen fernes Ende geschlossen ist, wirft eine `SocketException` („Broken pipe“).

Zu einer Exception sollte es nicht kommen. Wenn beide Kommunikationspartner das Protokoll korrekt implementieren, sind sie sich über das Ende der Kommunikation einig und schließen die Verbindung an beiden Enden zur gleichen Zeit. Mit zusätzlichen Methoden können Streams auch einseitig beendet werden, ohne den Socket zu schließen (siehe Seite 482).

7.2.5 Pufferung

Gepufferter Datentransport

Sender und Empfänger laufen auf verschiedenen Rechnern und arbeiten in der Regel nicht genau im Takt.

- Wenn der Empfänger langsamer liest, als der Sender schreibt, puffert das Netzwerk eine gewisse Menge Daten. Die Größe des Puffers hängt von vielen Faktoren ab, unter anderem von den Betriebssystemen und der Ausstattung der beteiligten Rechner und deren Konfiguration. Wenn der Puffer voll ist, blockieren schreibende Methodenaufrufe des Senders.
- Wenn der Empfänger schneller liest, als der Empfänger schreibt, blockieren die lesenden Aufrufe, bis Daten eintreffen.

Protokolle erfordern oft Leeren der Puffer

In der Regel fließen Nachrichten zwischen den Kommunikationspartnern gemäß Protokoll hin und her. Dazu muss eine Nachricht erst komplett übertragen sein, bevor die nächste formuliert werden kann. Wie alle Streams sind auch Socket-Streams aus Effizienzgründen gepuffert (siehe 1.2.3). Daher sind `flush`-Aufrufe am Ende ausgehender Nachrichten *unerlässlich*, um die Daten dem Empfänger zuverlässig zuzustellen.

Das trifft bei Netzwerkkommunikation in besonderem Maße zu, weil, anders als bei typischer Dateiein- und -ausgabe, ständig Daten über *dieselbe* Verbindung abwechselnd in beiden Richtungen geschickt werden. Ein konkretes Beispiel folgt im nächsten Abschnitt.

7.2.6 Beispiel: Echo-Client

Kommunikation mit Echo-Dienst

Der Echo-Service verwendet ein einfaches Protokoll (Seite 461). Das folgende Programm implementiert einen Echo-Client, der Verbindung zu Port 7 eines Servers aufnimmt. Der Name des Servers wird als erstes Kommandozeilenargument angegeben. Das Programm schickt weitere Kommandozeilenargumente, außer dem ersten, zum Server und gibt dessen Antworten aus.

```
import java.io.*;
import java.net.*;

public class EchoClient {
    public static void main(String... args) throws IOException {
        String hostname = args[0];
        int port = 7;   // well-known-Port des Echo-Service
        try(Socket socket = new Socket(hostname, port);
            OutputStream output = socket.getOutputStream();
            Writer writer = new OutputStreamWriter(output);
            PrintWriter pwriter = new PrintWriter(writer);
            InputStream input = socket.getInputStream();
            Reader reader = new InputStreamReader(input);
            BufferedReader breader = new BufferedReader(reader)) {
            for(int a = 1; a < args.length; a++) {
                pwriter.println(args[a]);
                pwriter.flush();
                System.out.println(breader.readLine());
            }
        }
    }
}
```

Listing 7.10: Kommandozeilenargumente zum Echo-Service schicken und Antworten ausgeben.

[39] Ein Dekorator kennt das dekorierte Objekt, aber nicht umgekehrt.

[40] Im Gegensatz zum Interface `AutoCloseable`, dessen `close`-Methode nicht idempotent sein muss (siehe Seite 38).

Der `flush`-Aufruf ist unverzichtbar! Ohne den Aufruf kommt höchstwahrscheinlich keine Antwort zurück, weil die Anfrage in einem Puffer stecken bleibt und nicht beim Server ankommt, der daraufhin auch keine Antwort schickt.

Beim Aufruf schickt ein Echo-Service die Kommandozeilenargumente wieder zurück:[41]

```
earth$ java EchoClient moon hello, anybody out there\?
hello,
anybody
out
there?
```

Dieses Beispiel nimmt an, dass der Rechner `moon` den Echo-Service auf dem *well-known* Port 7 anbietet.

▶ Viele Beispiele in diesem Kapitel beziehen sich auf ein Netzwerk mit zwei Rechnern namens `earth` und `moon`. Dabei nimmt `earth` in der Regel die Rolle eines Clients ein, während `moon` als Server arbeitet. Bei Aufrufbeispielen steht der Rechnername vor dem Promptzeichen `$`, wie beispielsweise in

```
earth$ hostname
earth
```

Das Unix-Kommando `hostname` gibt den Hostnamen des Rechners selbst aus, hier eben `earth`. ◀

7.2.7 Telnet-Client

Das Dienstprogramm „Telnet“ ist seit langer Zeit fester Bestandteil von Unix-Systemen und wurde seither auf fast alle Systeme portiert.[42] Der ursprüngliche Zweck war das Login (Öffnen von Eingabeaufforderungen) auf vernetzten Rechnern, allerdings ist diese Anwendung heute praktisch bedeutungslos.[43] Telnet kann aber mehr:

Testwerkzeug für textbasierte Netzwerkprotokolle

[41] Der Backslash in der Kommandozeile ist nötig, weil das Fragezeichen ein Meta-Symbol der Unix-Shell ist. Ohne Backslash interpretiert die Shell das Fragezeichen möglicherweise auf eigene Art.

[42] Auf Windows-Versionen ab Vista ist der Telnet-Client zwar installiert, aber in der Voreinstellung versteckt. Eine Anleitung zur „Aktivierung des Telnet-Clients unter Windows“ finden Sie im Internet.

[43] Telnet entstand in einer Zeit, in der die vernetzten Benutzer gutartig und Absicherungen unnötig waren. Demzufolge kennt es keine Schutzmechanismen. Außerhalb von geschlossenen und sicheren Umgebungen ist der Einsatz von Telnet zum Login heute fahrlässig.

Es dient als universeller Client, mit dem beliebige Ports vernetzter Rechner geöffnet und Protokolle interaktiv abgewickelt werden können.[44]

Test eines Echo-Service

Das folgende Beispiel zeigt den Dialog mit dem Echo-Service (Benutzereingaben sind unterstrichen) auf einem Rechner namens `moon`:[45]

```
$ telnet moon 7
Trying 192.168.1.24...
Connected to moon.
Escape character is '^]'.
hello?
hello?
anybody out there?
anybody out there?
^]
telnet> quit
Connection closed.
```

Die ersten Zeilen bedeuten nacheinander:

`Trying 192.168.1.24`
: Die numerische IP-Adresse des Servers, mit dem Telnet Verbindung aufnimmt.

`Connected to moon`
: Bestätigung, dass der Server die Verbindung auf dem gewählten Port akzeptiert.

`Escape character is '^]'`
: Hinweis auf die Tastenkombination, mit der der Dialog beendet werden kann. Die Schreibweise `^]` steht für Control + schließende, eckige Klammer.[46]

Test eines Time-Service

Im nächsten Beispiel antwortet der Time-Service eines Servers (vergleiche Seite 472). Der Server schickt sofort die Zeitangabe und beendet dann die Verbindung wieder. Auf eine Anfrage des Benutzers wartet der Server nicht.

```
$ telnet time.nist.gov 13
Trying 192.43.244.18...
Connected to time.nist.gov.
```

[44] Allerdings eignet sich Telnet nur für textbasierte Protokolle. Binärdaten können im Dialog nicht sinnvoll eingegeben und ausgelesen werden.

[45] Im Dialog mit Telnet ist es nicht nötig, das Fragezeichen zu entwerten, weil keine Shell im Spiel ist.

[46] Diese Zeichenfolge wurde für diesen Zweck gewählt, weil sie höchst unwahrscheinlich Teil eines Protokolls ist.

```
Escape character is '^]'.
55886 11-11-21 02:28:25 00 0 0 351.5 UTC(NIST) *
Connection closed by foreign host.
```

Ein Telnet-Client erweist sich als flexibles Testwerkzeug für Netzwerkserver.

7.2.8 Serversockets

Die bisher entwickelten Programme `PrintTextResponse` (Listing 7.9) und `EchoClient` (Listing 7.10) nehmen aktiv Verbindung auf, sind also Clients im Sinne der Client-Server-Architektur (Seite 462). Die dabei verwendeten Sockets vom Java-Typ `Socket` werden deshalb auch als „Client-Sockets“ bezeichnet. Client-Sockets dienen mittelbar dem Datenaustausch über die Streams, die sie bereitstellen.

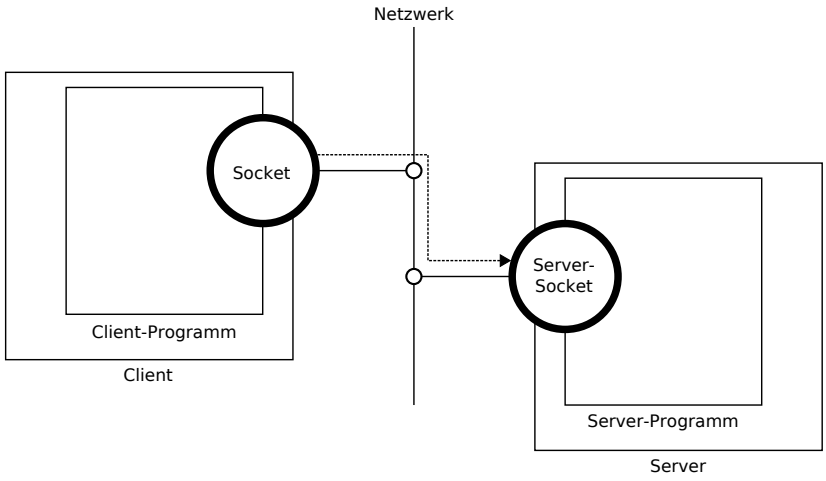
Serverprogramm öffnet passiven Serversocket

Ein Java-Programm, das als Server arbeitet, öffnet eine andere Art von Socket, nämlich einen „Serversocket“ vom Typ `ServerSocket`. Ein Serversocket ist passiv und wartet auf eingehende Anfragen. Der Konstruktor braucht keinen Hostnamen, sondern nur eine Portnummer:

```
ServerSocket serverSocket = new ServerSocket(port);
```

`ServerSocket`-Objekte funktionieren ganz anders als normale (Client-)`Socket`-Objekte. Die wichtigste `ServerSocket`-Methode ist `accept`. Ein Aufruf von `accept` blockiert so lange, bis ein Client-Request eintrifft.

Methode `accept` blockiert bis Client-Request

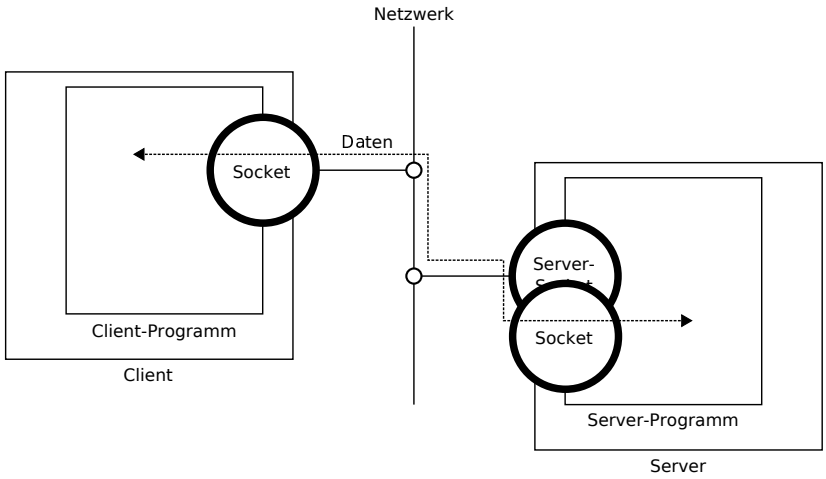


Dann spaltet `accept` auf Serverseite einen normalen Client-Socket ab und liefert ihn als Ergebnis zurück.

```
Socket socket = serverSocket.accept();
```

`accept` liefert normalen Socket zum Datenaustausch

Dieser neue Socket ist direkt mit seinem Gegenstück auf Clientseite verbunden und wird ganz genauso verwendet. Der Serversocket ist an der weiteren Kommunikation nicht mehr beteiligt:



Aufbau asymmetrisch, Verbindung symmetrisch

Sobald die Verbindung zwischen den zwei `Socket`-Objekten im Client und im Server arrangiert ist, arbeitet sie vollkommen symmetrisch. Es gibt keinen Unterschied mehr zwischen Client und Server. Den weiteren Verlauf der Kommunikation bestimmt alleine das Protokoll.

Asymmetrisch ist nur der Verbindungsaufbau: Ein Client öffnet auf seiner Seite einen `Socket`. Zu diesem Zeitpunkt muss der Server bereits einen `ServerSocket` geöffnet und die Methode `accept` aufgerufen haben. Wenn das nicht der Fall ist, kommt kein Kontakt zustande.

Schließen der Client-Sockets nach Protokollende

Nachdem der Server das Protokoll mit dem Client abgewickelt hat, schließen beide ihre Client-Sockets. Der Server wartet dann mit einem neuen `accept`-Aufruf auf den nächsten Client. Der `ServerSocket` bleibt für die gesamte Laufzeit des Servers offen. Erst wenn der Server endet, schließt er den `ServerSocket`.

Das folgende minimale Serverprogramm zeigt die Arbeitsweise:

```
import java.io.*;
import java.net.*;

public class NullServer {
    public static void main(String... args) throws IOException {
        int port = Integer.parseInt(args[0]);
```

```
        try(ServerSocket serverSocket = new ServerSocket(port)) {
            while(true)
                try(Socket s = serverSocket.accept();
                    InputStream input = s.getInputStream();
                    OutputStream output = s.getOutputStream()) {
                    // Protokoll abwickeln
                }
                catch(IOException iox) {
                    // Fehler
                }
        }
    }
}
```

Listing 7.11: Rumpf eines Server-Programmes.

Es korrespondiert mit dem minimalen Client-Programm `NullClient` (Listing 7.8).

7.2.9 Beispiele: Time- und Echo-Server

Eigener Time-Server

Auf der Grundlage von `NullServer` (Listing 7.11) lässt sich mit wenig Aufwand ein `TimeServer` definieren. Er erwartet keine Anfrage und öffnet daher keinen `InputStream`. Die `toString`-Methode eines neuen `Date`-Objekts liefert die Textform der aktuellen Serverzeit, die als Bytefolge über den `OutputStream` als Antwort zum Client geschickt wird. Damit endet die Kommunikation. Der Server ist bereit für den nächsten Request.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class TimeServer {
    public static void main(String... args) throws IOException {
        int port = Integer.parseInt(args[0]);
        try(ServerSocket serverSocket = new ServerSocket(port)) {
            while(true)
                try(Socket socket = serverSocket.accept();
                    OutputStream output = socket.getOutputStream()) {
                    output.write(new Date().toString().getBytes());
                    output.flush();
                }
                catch(IOException iox) {
                    // Fehler
                }
        }
    }
}
```

Listing 7.12: Server für einen Time-Service auf einem wählbaren Port.

Der Aufruf des Programms auf dem Server mit einem Port ab 1024 blockiert, weil der Server im `accept`-Aufruf auf Requests wartet:

```
moon$ java TimeServer 2000
```

Von einem Client aus kann jetzt der Dienst in Anspruch genommen werden:

```
earth$ java PrintTextResponse moon 2000
Mon Nov 21 14:34:58 CET 2011
earth$ java PrintTextResponse moon 2000
Mon Nov 21 14:35:01 CET 2011
```

Der Server akzeptiert Anfragen von jedem Programm, das das richtige Protokoll spricht. Auch der Zugriff mit Telnet liefert das entsprechende Ergebnis:

```
earth$ telnet moon 2000
Trying 192.168.1.24...
Connected to moon.
Escape character is '^]'.
Mon Nov 21 14:40:29 CET 2011
Connection closed by foreign host.
```

Administratorrechte für Dienst auf *priviledged* Port

Der *well-known* Port des Time-Service ist 13, einer der *priviledged* Ports. Um den Dienst auf diesem Port anzubieten, sind Administratorrechte erforderlich. Der Start als normaler Benutzer ohne ausreichende Rechte wird verweigert:

```
moon$ java TimeServer 13
Exception in thread "main" java.net.BindException: Permission denied
```

Eigener Echo-Server

Auch ein Echo-Server lässt sich mit wenig Aufwand implementieren:

```
import java.io.*;
import java.net.*;

public class EchoServer {
    public static void main(String... args) throws IOException {
        int port = Integer.parseInt(args[0]);
        try(ServerSocket serverSocket = new ServerSocket(port)) {
            while(true)
                try(Socket socket = serverSocket.accept();
                    InputStream input = socket.getInputStream();
```

```
                    OutputStream output = socket.getOutputStream()) {
                    int b = input.read();
                    while(b >= 0) {
                        output.write(b);
                        output.flush();
                        b = input.read();
                    }
                }
                catch(IOException iox) {
                    // Fehler
                }
            }
        }
}
```

Listing 7.13: Echo-Server auf einem wählbaren Port.

Wiederum erfordert der Start auf dem *well-known* Port 7 entsprechende Rechte.

7.2.10 Verbindung auf einzelnen Netzwerkschnittstellen

Dienste auf einzelnen Netzwerkschnittstellen

Der `TimeServer` (Listing 7.12) antwortet auf jede Anfrage. Vertraulich ist die Systemzeit des Servers nicht, daher zieht dieser Dienst auch kein datenschutzrechtliches Problem nach sich. Andere Dienste sollen aber nicht unbedingt allen Clients zur Verfügung stehen. Passwortgesicherte, verschlüsselte Verbindungen (Seite 485) lösen dieses Problem gründlich, erfordern aber zusätzlichen Code. Ein oft ausreichender Schutz lässt sich auch mit weniger Aufwand erreichen.

Die meisten Rechner sind mit verschiedenen Netzwerken verbunden. Das Programm `IPAddresses` (Listing 7.3) zeigt zum Beispiel, dass dieser Rechner Host in vier Netzwerken ist:

```
$ java IPAddresses
eth0    /129.187.208.11 (24)
lo      /127.0.0.1 (0)
tap0    /10.4.0.1 (24)
wlan0   /192.168.1.23 (24)
```

Es könnte zum Beispiel sinnvoll sein, einen bestimmten Dienst nur im privaten Netzwerk `192.168.1.0/24` anzubieten und nicht im öffentlichen Netzwerk `129.187.208.0/24`, das letztlich mit dem ganzen Internet in Kontakt stehen könnte.

Überladener `ServerSocket`-Konstruktor

Der mit drei Parametern überladene `ServerSocket`-Konstruktor

```
ServerSocket(port, backlog, address)
```

erwartet neben dem Port und einem zweiten Argument, das hier keine Rolle spielt und mit 0 angegeben werden kann, noch eine IP-Adresse als drittes Argument. Der so erzeugte Serversocket nimmt nur Verbindungen auf dieser Adresse an. Die anderen Addressen des Rechners werden ignoriert. Die Adresse wird als Objekt vom Typ `InetAddress` übergeben. Eine `InetAddress` kann auf verschiedenen Wegen erzeugt werden, wie zum Beispiel:

```
InetAddress.getByName("localhost")
InetAddress.getByName("192.168.1.23")
NetworkInterface.getByName("wlan0").getInetAddresses().nextElement()
```

Time-Server für ausgewähltes Netzwerk

Der folgende Server, eine Variation des `TimeServer` (Listing 7.12), erwartet auf der Kommandozeile einen Port und den Namen einer Netzwerkschnittstelle, auf der er Requests entgegennimmt:

```
import java.io.*;
import java.net.*;
import java.util.*;

public class TimeServerLtd {
    public static void main(String... args) throws IOException {
        int port = Integer.parseInt(args[0]);
        NetworkInterface adapter = NetworkInterface.getByName(args[1]);
        InetAddress address = adapter.getInetAddresses().nextElement();
        try(ServerSocket serverSocket = new ServerSocket(port, 0, address)) {
            // identisch mit TimeServer ...
        }
    }
}
```

Listing 7.14: Server für einen Time-Service auf einer bestimmten Netzwerkschnittstelle.

Beim Start mit

```
$ java TimeServerLtd 2000 wlan0
```

liefert er die Systemzeit in das Netzwerk, das mit der Schnittstelle `wlan0` verbunden ist. Anfragen aus anderen Netzwerken werden abgewiesen.

7.2.11 Teilweiser Abbau

Schließen eines Streams ohne Schließen des Sockets

Ein Aufruf von `close` schließt immer einen Socket und seine beiden Streams gemeinsam (Seite 472). Die beiden `Socket`-Methoden

```
void  shutdownInput()

void  shutdownOutput()
```

beenden nur einen der beiden Streams.[47] Sie schließen nicht den Socket und insbesondere auch nicht den jeweils anderen Stream, der normal weiterbenutzt werden kann.

Diese Methoden sind nützlich, wenn ein Protokoll die Übertragung beliebiger Daten vorsieht und kein Fluchtwert oder anderes Signal zur Begrenzung existiert.

Protokoll ohne explizites Ende

Im Protokoll der folgenden Beispielprogramme schickt der Client zuerst ein Betriebssystem-Kommando als String zum Server und beendet daraufhin den Stream. Der Server führt das Kommando bei sich aus, schickt die Kommandoausgabe zurück und beendet dann die Kommunikation. Dieses Protokoll demonstriert den teilweisen Abbau der Verbindung.[48]

Das Client-Programm erwartet einen Hostnamen, eine Portnummer und ein Kommando auf der Kommandozeile. Es schickt die Bytedarstellung des Kommandos zum Server und beendet den ausgehenden Stream mit `shutdownOutput`. Dann liest es die Antwort des Servers und schreibt sie auf den Bildschirm.

```
import java.io.*;
import java.net.*;

public class RemoteCommandClient {
    public static void main(String... args) throws IOException {
        String hostname = args[0];
        int port = Integer.parseInt(args[1]);
        try(Socket socket = new Socket(hostname, port);
            InputStream input = socket.getInputStream();
            OutputStream output = socket.getOutputStream()) {
            for(int a = 2; a < args.length; a++) {
                output.write(args[a].getBytes());
                output.write('\n');
            }
            socket.shutdownOutput();
            Streams.copy(input, System.out);
        }
    }
}
```

Listing 7.15: Client, der ein Kommando zu einem Server schickt und die Antwort ausgibt.

[47] Die beiden Methoden rufen nicht unter der Hand doch wieder `close` auf, sondern versetzen den Stream in einen Zustand, der für das andere Ende wie ein geschlossener Stream aussieht.

[48] Man hätte das Protokoll auch anders entwerfen und zum Beispiel das Ende des Kommandos durch einen Zeilenwechsel anzeigen können.

▶ Die statische Methode `copy` der Hilfsklasse `Streams` kopiert einen kompletten `InputStream` auf einen `OutputStream`:

```
import java.io.*;

public class Streams {
    public static void copy(final InputStream from, final OutputStream to) throws IOExcep
        for(int b = from.read(); b >= 0; b = from.read())
            to.write(b);
        to.flush();
    }
}
```

Listing 7.16: Hilfsklasse mit einer statischen Methode, die einen Inputstream byteweise auf einen Outputstream kopiert.

Eine effizientere Version könnte einen Byte-Puffer verwenden. Effizienz steht hier allerdings nicht im Vordergrund. ◀

Das entsprechende Serverprogramm liest Bytes vom Socket, bis der Stream endet. Es erzeugt einen String aus den Bytes und führt das entsprechende Kommando aus. Die Ausgabe des Kommandos geht zurück an den Client.

```
import java.io.*;
import java.net.*;

public class RemoteCommandServer {
    public static void main(String... args) throws IOException, InterruptedException {
        int port = Integer.parseInt(args[0]);
        try(ServerSocket serverSocket = new ServerSocket(port)) {
            while(true)
                try(Socket socket = serverSocket.accept();
                    InputStream input = socket.getInputStream();
                    OutputStream output = socket.getOutputStream()) {
                    ByteArrayOutputStream bytesOutput = new ByteArrayOutputStream();
                    Streams.copy(input, bytesOutput);
                    Process process = new ProcessBuilder(bytesOutput.toString().split("\\
                    .redirectErrorStream(true)
                    .start();
                    process.waitFor();
                    Streams.copy(process.getInputStream(), output);
                }
                catch(IOException iox) {
                    iox.printStackTrace();
                }
        }
    }
}
```

Listing 7.17: Server, der beliebige Kommandos ausführt.

Fernsteuerung des Servers

Diese Programme erlauben den Aufruf von Shell-Kommandos über das Netzwerk.

Im folgenden Beispiel läuft `RemoteCommandServer` auf dem Server `moon` unter Windows. Ein Aufruf des Clients auf einem anderen Rechner liefert die Server-Ausgabe des Kommandos:

```
earth$ java RemoteCommandClient moon 12345 cmd /c dir
 Datenträger in Laufwerk Z: ist VBOX_tmp
 Volumeseriennummer: 0000-000F

 Verzeichnis von Z:\.netbeans\07-networking\build\classes

22.11.2011  08:47    <DIR>          ch03http
22.11.2011  08:47    <DIR>          ch02sockets
22.11.2011  08:47    <DIR>          ch01basics
               0 Datei(en)            620 Bytes
               3 Verzeichnis(se),  1.290.899.456 Bytes frei
```

Dieser Server führt *jedes* Kommando aus, das man ihm schickt. Starten Sie ihn nur in einem geschützten Netzwerk!

7.2.12 Verschlüsselte Verbindung

Anforderungen an vertrauliche Kommunikation

Das Programm `RemoteCommandServer` (Listing 7.17) ist ein Beispiel für einen Server, das nicht unkontrolliert benutzt werden darf. Dabei sind die folgenden Einschränkungen nötig:[49]

1. Der Server darf nur Clients bedienen, die dazu berechtigt sind. Dazu müssen Clients ihre Identität glaubhaft nachweisen, sie müssen sich authentisieren.
2. Der Netzwerkverkehr zwischen dem Server und berechtigten Clients darf nicht von Dritten beobachtet oder gar verändert werden. Die Verbindung muss vertraulich und manipulationssicher sein.

Diese Aspekte fallen unter das allgemeine Thema „IT-Sicherheit“, das in Java gut unterstützt ist. Allerdings ist IT-Sicherheit ein weitläufiges Gebiet mit rasch wachsender Bedeutung. Eine ausführliche Diskussion sprengt den Rahmen dieses Textes, daher muss es bei einem verhältnismäßig einfachen Beispiel bleiben.

Verschlüsselte Byteströme

Netzwerkkommunikation beruht auf Byteströmen, die in Kapitel 1 genauer be-

[49] Eine weitere Anforderung, auf die hier nicht weiter eingegangen wird, ist die ausreichende Autorisierung. Dabei feht es um die Frage, ob der Client überhaupt berechtigt ist, einen Dienst in Anspruch zu nehmen.

schrieben sind. Dabei bieten Filterklassen (Kapitel 1.4) einen flexiblen Mechanismus, um existierende Streams mit zusätzlichen Fähigkeiten auszustatten. Die beiden Klassen `CipherInputStream` und `CipherOutputStream` sind Filter, die einen anderen Stream ver- beziehungsweise entschlüsseln.

Auswahl an Verschlüsselungsalgorithmen

Cipherstreams brauchen neben einem zugrunde liegenden Stream auch einen Verschlüsselungsalgorithmus. Die Java-Bibliothek stellt eine ganze Auswahl mit unterschiedlichen Eigenschaften zur Verfügung. Die Wahl fällt hier auf einen „symmetrischen Algorithmus“, der mit einem Passwort verschlüsselt und mit dem gleichen Passwort entschlüsselt.[50] Der hier gewählte Verschlüsselungsalgorithmus heißt AES (*Advanced Encryption Standard*). Die statische Factory-Methode `Cipher.getInstance` erzeugt ein Objekt, das den Algorithmus repräsentiert.

```
Cipher cipher = Cipher.getInstance("AES");
```

Vorbereitung der Verschlüsselung

In dieser Form ist der Algorithmus noch nicht bereit. Die Methode `init` initialisiert ihn mit zwei Argumenten:

```
void  init(int opmode, SecretKey secretKey)
```

1. Der Parameter `opmode` legt mit einer der beiden Konstanten `Cipher.ENCRYPT_MODE` und `Cipher.DECRYPT_MODE` die Arbeitsrichtung (Ver- oder Entschlüsseln) fest.
2. `secret` enthält das Passwort in einer Form, mit der der Verschlüsselungsalgorithmus arbeiten kann. Dazu muss es in ein Objekt des Typs `SecretKey` verpackt werden.
   Zuerst werden das Passwort und ein Array von acht Bytes zusammengepackt.[51] Das Array macht es einem mitlauschenden Angreifer schwerer, das Passwort zu erraten. Hier wird es aus einem String gewonnen. Noch besser wäre ein zufälliger Inhalt.[52] Das dritte Argument (1024) ist ein Rundenzähler, der ebenfalls die Qualität des später erzeugten Schlüssels beeinflusst. Er sollte wenigstens bei 1000 liegen. Der letzte Argument legt die Bitlänge des Schlüssels fest. Längere Schlüssel sind schwerer zu knacken. Derzeit gilt ein AES-Schlüssel mit 128 Bit Länge als akzeptabel, einer mit 256 Bit als sicher.

   ```
   KeySpec keySpec = new PBEKeySpec(password.toCharArray(),
                                    "donttell".getBytes(), 1024, 256);
   ```

[50] Im Gegensatz dazu verwenden „asymmetrische Algorithmen“ einen geteilten Schlüssel, mit einer Hälfte zum Verschlüsseln und der anderen Hälfte zum Entschlüsseln.

[51] Diese Beigabe wird als „Salt“ bezeichnet und muss beim Ver- und Entschlüsseln gleich sein.

[52] Aus Sicherheitsgründen akzeptiert der `PBEKeySpec`-Konstruktor keine Strings, die als Literale unter Umständen allzu leicht im Bytecode zu finden wären. Außerdem kann der Inhalt eines Arrays nach Gebrauch gelöscht werden, während das bei einem String nicht möglich ist.

Das eigentliche `SecretKey`-Objekt produziert eine „Factory“ vom Typ `SecretKeyFactory`, die aber verschiedene Arten von Schlüsseln erzeugen können. Das Argument `"PBKDF2WithHmacSHA1"` wählt die hier benötigte Art aus:

```
SecretKeyFactory factory =
   SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
```

Der letzte Schritt erzeugt mithilfe der Factory aus der `KeySpec` das gewünschte `SecretKey`-Objekt:

```
SecretKey secretKey =
   new SecretKeySpec(factory.generateSecret(keySpec).getEncoded(), "AES");
```

Kommandoserver mit gesicherter Kommunikation

Der folgende Server erweitert `RemoteCommandServer` (Listing 7.17) um die Konstruktion zweier `Cipher`-Objekte, von denen eines zum Entschlüsseln des Requests und eines zum Verschlüsseln des Responses dient. Er initialisiert die `Cipher`-Objekte in jedem Request-Response-Zyklus neu. Das Programm erwartet auf der Kommandozeile eine Portnummer und eine Passwort.

```
import java.io.*;
import java.net.*;
import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class RemoteAESPasswordCommandServer {
    public static void main(String... args) throws GeneralSecurityException, IOException
        int port = Integer.parseInt(args[0]);
        String password = args[1];

        // Ver- und Entschlüsseler bereitstellen
        String algorithm = "AES";
        SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
        KeySpec keySpec = new PBEKeySpec(password.toCharArray(), "donttell".getBytes(), 1
        SecretKey secretKey = factory.generateSecret(keySpec);
        secretKey = new SecretKeySpec(secretKey.getEncoded(), algorithm);
        Cipher encoder = Cipher.getInstance(algorithm);
        Cipher decoder = Cipher.getInstance(algorithm);

        try(ServerSocket serverSocket = new ServerSocket(port)) {
            while(true)
                try(Socket socket = serverSocket.accept();
                    InputStream input = new CipherInputStream(socket.getInputStream(), de
                    OutputStream output = new CipherOutputStream(socket.getOutputStream()
                    // Ver- und Entschlüsseler initialisieren
                    encoder.init(Cipher.ENCRYPT_MODE, secretKey);
                    decoder.init(Cipher.DECRYPT_MODE, secretKey);
                    // Request empfangen
                    ByteArrayOutputStream bytesOutput = new ByteArrayOutputStream();
                    Streams.copy(input, bytesOutput);
                    String command = bytesOutput.toString().trim();
```

```
                    System.out.println("received: " + command);
                    // Kommando ausführen
                    Process process = new ProcessBuilder(command.split("\\s+")).redirectE
                    process.waitFor();
                    // Response senden
                    Streams.copy(process.getInputStream(), output);
                }
                catch(InterruptedException | IOException ex) {
                    ex.printStackTrace();
                }
            }
        }
    }
```

Listing 7.18: Server, der beliebige Kommandos ausführt und über eine verschlüsselte Verbindung kommuniziert.

Nach dem Start auf dem Host `moon` wartet der Server auf eingehende Requests:

```
moon$ java RemoteAESPasswordCommandServer 4000 topsecret
```

Der Client baut die `Cipher`-Objekte ganz genauso auf. Er erwartet als Kommandozeilenargumente den Namen des Servers, die Portnummer, das vereinbarte Passwort und ein Betriebssystem-Kommando.

```
import java.io.*;
import java.net.*;
import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class RemoteAESPasswordCommandClient {
    public static void main(String... args) throws IOException, GeneralSecurityException
        String hostname = args[0];
        int port = Integer.parseInt(args[1]);
        String password = args[2];

        // Ver- und Entschlüsseler bereitstellen, wie Server ...
        try(Socket socket = new Socket(hostname, port);
            // Streams initialisieren, wie Server ...) {
            // encoder und decoder initialisieren, wie Server ...
            // Kommandozeile lesen und als Request senden
            for(int a = 3; a < args.length; a++) {
                output.write(args[a].getBytes());
                output.write(' ');
            }
            // Block auffüllen
            for(int i = 0; i < 32; i++)
                output.write(' ');
            socket.shutdownOutput();
```

```
            // Response empfangen und ausgeben
            Streams.copy(input, System.out);
        }
    }
}
```

Listing 7.19: Client, der Kommandos über eine verschlüsselte Verbindung zum Server schickt.

Geblockte Arbeitsweise des Verschlüsselungsalgorithmus

Beim Senden des verschlüsselten Kommandos ergibt sich ein zusätzliches Problem aus der Arbeitsweise des Algorithmus. Er verarbeitet keine einzelnen Bits und Bytes, sondern immer ganze Blöcke. Ein Aufruf von `close` eines `CipherOutputStream` füllt unvollständige Blöcke auf und schickt die Restdaten auf die Reise, nicht aber `flush` und auch nicht `shutdownOutput`. Gerade `close` kann der Client *nicht* aufrufen, nachdem das Kommando zum Server geschickt wurde, weil sonst auch der Socket geschlossen und die Verbindung abgerissen wäre. Dieses Problem lässt sich mit einem „work around“ lösen: Der Client schiebt nach dem Kommando noch eine Anzahl Leerzeichen nach, die nur dafür sorgen, dass alle Nutzdaten übermittelt werden.[53]

Jeder Client, der das geheime Passwort kennt, kann nun Kommandos auf dem Server ausführen.

```
earth$ java RemoteAESPasswordCommandClient moon 4000 topsecret ls -l
total 0
drwx------ 2 rs users 120 Nov 28 08:23 ch01basics
drwx------ 3 rs users 340 Nov 28 08:23 ch02sockets
drwx------ 2 rs users 300 Nov 28 08:23 ch03http
```

▶ Möglicherweise bricht Ihr Server beim ersten Kontaktversuch eines Clients mit einer `InvalidKeyException` ab. Die Ursache liegt in der Java-Installation, die in der Voreinstellung höchstens 128 Bit lange Schlüssel erlaubt. Das hat politische Gründe, weil manche Länder beliebig lange (und damit sehr sichere) Schlüssel nicht wünschen. Der Aufruf

```
Cipher.getMaxAllowedKeyLength("AES")
```

liefert die maximale Schlüssellänge Ihres JDK.

Der Fehler verschwindet, wenn Sie in den Programmen `RemoteAESPasswordCommandClient` (Listing 7.19) und `RemoteAESPasswordCommandServer` (Listing 7.18) die Schlüssellänge auf 128 Bit verringern.

[53] Es muss angemerkt werden, dass das kein allgemeines Problem von Cipherstreams ist, sondern ein spezifisches Problem dieser Anwendung in Kombination mit dem hier gewählten Verschlüsselungsalgorithmus.

Die Begrenzung auf 128 Bit lässt sich auch ganz beseitigen. Laden Sie dazu von der Downloadseite des Oracle JDK unter dem Eintrag „Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files 7“ die Zip-Datei `UnlimitedJCEPolicyJDK7.zip` herunter. Sie enthält unter anderem die zwei Jar-Dateien `US_export_policy.jar` und `local_policy.jar`, die Sie in das Subdirectory `jre/lib/security` Ihrer Java-Installation kopieren. Die dort bereits vorhandenen Dateien gleichen Namens werden dabei ersetzt.

In Deutschland ist dieses Vorgehen legal. ◀

In diesem Beispiel organisieren Client und Server, also die Anwendungsprogramme, selbst die Verschlüsselung. Man spricht daher von *application level security*. Dem gegenüber steht *link level security*, wo die Verbindung selbst auf Ebene 4 oder tiefer pauschal und ohne Zutun einzelner Anwendungen abgesichert ist.

7.3 HTML, HTTP und Webserver

7.3.1 Idee

Trennung von Inhalt, Struktur und Darstellung

Die Auszeichnungssprache HTML (*Hypertext Markup Language*) diente ursprünglich zur Formatierung und zum Austausch wissenschaftlicher Arbeiten. Heute hat sich HTML über mehrere Versionen hinweg weit darüber hinaus entwickelt und ist überaus erfolgreich. Wesentliche Merkmale von HTML sind

1. Die Trennung von
   - Inhalt
   - Struktur und
   - Präsentation

   Darstellung Sache des Lesers

   Dabei fixiert der Autor Inhalt und Struktur, überlässt die Präsentation aber dem Leser. Technisch manifestiert sich
   - der Inhalt in Fließtext,
   - die Struktur in Markierungen im Text und
   - die Präsentation im Browser, der eine HTML-Seite darstellt.
2. Eine verhältnismäßig einfache und standardisierte Markierungssprache in Textform.
3. Die Verknüpfung verschiedener HTML-Seiten durch Hyperlinks, die eine einfache Navigation erlauben.

7.3.2 Webserver und Browser

Rollen von Webserver und Webbrowser

Webserver stellen HTML-Dateien zur Verfügung, die im einfachsten Fall im Filesystem des Servers gespeichert sind. Diese HTML-Dateien regeln nur Inhalt und Struktur.[54] Die Rolle der Clients nehmen **Webbrowser** ein. Sie fordern Dokumente an, analysieren Inhalt und Struktur und präsentieren sie schließlich dem Benutzer.

HTTP als gemeinsames Protokoll

Das gemeinsame Protokoll zwischen Webserver und Browser ist **HTTP** (*Hypertext Transfer Protocol*[55]; siehe Seite 461). HTTP eignet sich aber nicht nur für HTML-Seiten, sondern für Dateien beliebigen Inhalts. Es ist Sache des Browsers, die empfangenen Dateien für den Benutzer sinnvoll darzustellen.

Abgesehen von der Darstellung von HTML-Seiten erlauben Browser das bequeme Folgen von **Hyperlinks**, die in den Seiten eingebettet sind und auf andere Seiten anderer Server verweisen.

7.3.3 URLs

URLs verweisen auf Ressourcen im Netzwerk

URLs (*Uniform Resource Locator*) sind global eindeutige Beschreibungen für Orte von „Ressourcen“, wie zum Beispiel HTML-Seiten und andere Dateien.[56] Sie sind folgendermaßen aufgebaut, wobei Bestandteile in eckigen Klammern optional sind:[57]

```
protocol://server[:port][/resource]
```

Dabei bedeuten:

protocol Das Protokoll, mit dem die Ressource erreicht werden kann. In der Regel ist das `http`. Die meisten Browser können auch mit einigen anderen Protokollen umgehen, wie beispielsweise `https` und `ftp`.[58]

[54] Ganz stimmt das heute nicht mehr. Zunehmend wollen Seiten-Autoren auf die Darstellung Einfluss nehmen und versuchen genaue Vorgaben zu machen, wie ein Browser bestimmte Strukturen darzustellen hat.

[55] HTTP ist dokumentiert auf `http://www.ietf.org/rfc/rfc2616.txt`.

[56] Allgemein lokalisieren URLs beliebige „Ressourcen“ in einem Netzwerk, wobei Dateien nur eine von mehreren Möglichkeiten sind. Weiter hinten in diesem Kapitel tauchen auch andere Ressourcen als Dateien auf.

[57] Tatsächlich sind URLs komplexer. Eine ausführliche Beschreibung gibt `http://tools.ietf.org/html/rfc1630`.

[58] Eine komplette Liste findet sich auf `http://www.iana.org/assignments/uri-schemes.html`.

server Der Hostname oder die IP-Adresse[59] des Servers, der die Datei zur Verfügung stellt.

port Die Portnummer zum Zugriff auf die Datei. Bei einer fehlenden Angabe fügt der Browser den *well-known* Port des Protokolls ein, also zum Beispiel 80 bei `http`.

resource Bezeichnung der Ressource, die wie ein Pfadname in einem Filesystem aufgebaut ist.[60] Der Server kann diesen String direkt als Pfadname in seinem Filesystem verwenden, kann ihn aber auch modifizieren oder ganz anders interpretieren. Wenn diese Angabe fehlt, ergänzen viele Webserver beim Protokoll `http` den Namen `/index.html`.

Die URL

```
http://dev.w3.org/html5/spec/Overview.html
```

benennt eine HTML-Seite, die

- über das Protokoll `http` geliefert wird,
- auf dem Server `dev.w3.org` liegt,
- über Port 80, den *well-known* Port für HTTP, erreicht werden kann und
- auf dem Server unter dem Pfad `html5/spec/Overview.html` zu finden ist.

▶ Neben dem Akronym URL (*Uniform Resource Locator*, manchmal auch als *Universal Resource Locator* bezeichnet) tauchen gelegentlich noch die Abkürzungen URN (*Uniform Resource Name*) und URI (*Uniform Resource Identifier*) auf.

- URLs legen einen bestimmten *Ort* fest, an dem eine gesuchte Ressource zu finden ist und wo sie abgeholt werden kann. Das Protokoll regelt außerdem, *wie* man an die Daten herankommt (`http`, `https`, `ftp` und so weiter).
- URNs sind *Namen* für Informationen ohne bestimmten Ort. Beispielsweise ist `tel:+49-89-1265-3717` ein URN für die Telefonnummer und `mailto:rs@cs.hm.edu` ein URN für die E-Mail-Adresse des Autors. Eine Telefonnummer und eine E-Mail-Adresse liegen an keinem bestimmten Ort und werden daher auch nicht mit *Locators* angesprochen.

„URI“ ist ein Sammelbegriff für URLs und URNs. In diesem Kapitel geht es nur um URLs, nicht um URNs und URIs. ◀

[59] Statt der üblichen Schreibweise mit vier durch Punkte getrennte Bytewerte kann auch eine einzelne Dezimalzahl angegeben werden, wie zum Beispiel `http://3500671255/index.html`.

[60] Der Schrägstrich als Trenner gilt für alle Systeme, unabhängig vom Betriebssystem, auf dem Server und Browser laufen.

7.3.4 HTML

Sprache zur Definition der Dokumentstruktur

HTML definiert Markierungen, die die Struktur eines Dokuments festlegen. Die Schreibweise der Markierungen ist an XML (*eXtensible Markup Language*) angelehnt (Kapitel 3), allerdings mit einigen Abweichungen im Detail.[61] Seit den Anfängen in den 1990er Jahren entwickelten sich mehrere HTML-Versionen. Heute maßgeblich sind die stabile HTML-Version 4[62] und die nächste Version 5[63], die in den weiteren Beispielen verwendet wird.

Elemente, Tags und Text

Wie XML-Dokumente bestehen HTML-Dokumente aus geschachtelten Elementen in der Schreibweise

```
<element>...</element>
```

Zwischen dem öffnenden Tag <*element*> und dem schließenden Tag </*element*> stehen weitere Elemente und Text, der den eigentlichen Inhalt des Dokuments ausmacht. Zum Beispiel markiert das Element `em` (*emphasized*) hervorgehobene Textpassagen, wie in:

```
Willkommen auf meiner <em>Homepage!</em> Das bin <em>ich</em> ...
```

Layout, wie Zeilenumbruch, Einrückung und Zwischenraum, ist im Allgemeinen nicht signifikant. Das vorhergehende Beispiel könnte auch geschrieben werden als

```
Willkommen auf meiner <em>
   Homepage!</em> Das
bin <em>ich</em> ...
```

Elemente können geschachtelt werden. Das Element `h1` (*header level 1*) markiert zum Beispiel eine Überschrift, wie im nächsten Beispiel:

```
<h1>Willkommen auf meiner <em>Homepage!</em></h1> Das bin <em>ich</em> ...
```

[61] HTML-Dokumente sind deshalb im Allgemeinen *keine* validen XML-Dokumente, sie sind nicht einmal wohlgeformt (Seite 185). Eine Ausnahme ist die HTML-Variante „XHTML“, die sich genau an die Syntax von XML hält (`http://www.w3.org/TR/xhtml1`). XHTML-Dokumente sind folglich valide XML-Dokumente. Die gängigen Browser kommen sowohl mit HTML als auch mit XHTML zurecht, allerdings ist XHTML weniger verbreitet als HTML.

[62] Spezifikation verfügbar auf `http://www.w3.org/TR/html4`.

[63] Letzter Stand auf `http://www.w3.org/TR/html5`.

Einige Elemente haben überhaupt keinen Inhalt. Bei diesen „leeren Elementen“ sollte das schließende Tag wegbleiben.[64]

Attribute mit Zusatzinformationen für Elemente

Einige Elemente erfordern Zusatzinformationen, die nicht zum Inhalt zählen und die der Browser folglich nicht als lesbaren Text darstellt. Solche Angaben werden als „Attribute“[65] bezeichnet und in die öffnenden Tags der betreffenden Elemente eingefügt. Jedes Attribut besteht aus einem Namen und einem Wert, die mit = getrennt sind. Der Wert kann in Gänsefüßchen oder Hochkommas gesetzt werden, wenn er Sonderzeichen oder Zwischenraum enthält.[66]

```
<element name=value ...>...
```

Ein Beispiel ist das Element `a` (*anchor*), das einen Link auf ein anderes Dokument darstellt. Das entsprechende Attribut heißt `href` (*hyper reference*), der Attributwert ist die URL des Zieldokuments. Im folgenden Beispiel verweist der Text „meine Freunde“ auf das Zieldokument `friends.html`.[67]

```
<a href=friends.html>meine Freunde</a>
```

Manche Elemente sind mit mehreren verschiedenen Attributen versehen, die nacheinander aufgezählt werden. Die Reihenfolge ist dabei ohne Bedeutung.

Ein Beispiel ist das Element `img` (*image*), das ein Bild in die Seite einfügt. Dieses Element hat zwar keinen Inhalt, aber zwei Attribute:

- `src` (*source*) nennt als Wert die URL[68] einer Bilddatei,
- `alt` (*alternate text*) gibt eine Beschreibung des Bildes als kurzen Text an. Dieser Text ist für Browser bestimmt, die keine Bilder darstellen können.

Das folgende Beispiel zeigt ein `img`-Element mit beiden Attributen. Der Wert von `alt` ist in Gänsefüßchen gesetzt, weil er Zwischenraum enthält. Bei `src` ist das nicht nötig.

```
<img src=photograph.png alt="Bild von mir">
```

[64] Das empfiehlt sich angesichts der Verbreitung von HTML 4 und 5. Die verschiedenen HTML-Versionen weichen hier voneinander ab: XHTML verlangt ganz XML-konform die Angabe eines schließenden Tags `<element/>`, die in HTML 5 optional und in HTML 4 sogar unzulässig ist.

[65] Der ganze Katalog steht auf `http://www.w3.org/TR/html4/index/attributes.html`.

[66] Manche HTML-Attribute, wie zum Beispiel `checked` und `ismap`, haben keinen Wert. In diesem Fall steht der Attributname alleine. In XML sind sowohl Attributwerte als auch Begrenzer Pflicht.

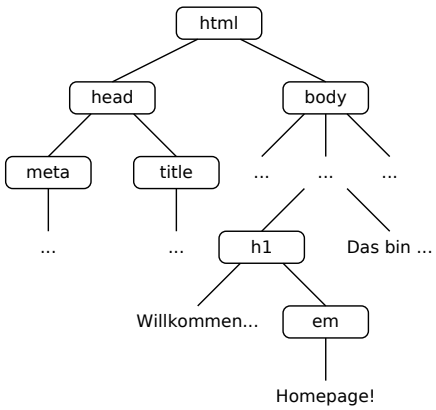
[67] Bei URLs ohne Protokoll, Server, Port und Pfad werden automatisch die Angaben des Dokuments, in dem der Link steht, übernommen.

[68] Wie bei `href`-Attributen von `a`-Elementen gilt, dass fehlende Angaben von Protokoll, Server, Port und Pfad von der einbettenden Seite kopiert werden.

7.3.5 DOM

Baum aus Objekten als interne Darstellung

HTML-Dokumente beschreiben eine Baumstruktur in Textform, ebenso wie XML-Dokumente. Dabei entsprechen die HTML-Elemente den inneren Knoten im Baum, die Texte und leeren Elemente den Blättern des Baums. Diese Baumstruktur wird als **DOM** (*Document Object Model*) bezeichnet. Die folgende Skizze zeigt ein Beispiel eines DOM:



Browser lesen HTML, konstruieren daraus ein DOM

Browser bauen aus einer HTML-Datei zunächst intern ein DOM auf und erzeugen aus dem DOM im zweiten Schritt eine grafische Darstellung. Ein Problem für Browser ist der Umgang mit nicht ganz korrekten HTML-Dateien, die in der Praxis bei Weitem in der Überzahl sind. Anders als ein Compiler oder XML-Parser kann ein Browser aber nicht beim ersten kleinsten Fehler abbrechen, sondern muss versuchen, trotz Ungereimtheiten eine sinnvolle Struktur herauszufinden. Ein solcher fehlertoleranter Parser ist schwerer zu schreiben als ein formal präzise arbeitender Parser.

7.3.6 Aufbau eines Dokuments

HTML fixiert Elemente und ihr Arrangement

HTML legt einen Katalog von Elementen mit Attributen fest und regelt die zulässigen Kombinationen.

- Das Wurzelelement, das das ganze HTML-Dokument umfasst, heißt `html`. `head` und `body` sind zulässige Kindelemente von `html`.
- `head` enthält Metadaten, das heißt Angaben *über* das Dokument. Dazu zählen die Elemente `meta` und `title`.
- `meta` legt unter anderem das Encoding des Dokuments fest (siehe Seite 84).

- `title` definiert einen Dokumenttitel, den die meisten Browser in der Titelzeile des Browserfensters einblenden.
- Der eigentliche Seiteninhalt folgt im Element `body`.

Einige Beispiele von Elementen im `body` sind:

- `div` (*division*) zum Eingrenzen von Abschnitten.
- `h1` (*header level 1*) für Überschriften auf oberster Ebene. Entsprechend markieren `h2`, `h3` und so weiter untergeordnete Überschriften.
- `em` (*emphasized*) für hervorgehobene Textpassagen.
- `img` (*image*) zum Einbetten von Bildern.
- `br` (*line break*) für Zeilenumbrüche. Wie oben erwähnt, spielt das Layout des HTML-Dokuments keine Rolle. Zeilenumbrüche müssen daher explizit verlangt werden, damit sie sichtbar werden.
- `a` (*anchor*) für Verweise auf andere Seiten und Ressourcen.

Das folgende Listing zeigt ein einfaches, aber komplettes HTML-Dokument.[69] In der ersten Zeile steht eine „Doctype-Deklaration“, die die verwendete HTML-Version festlegt. Die Doctype-Deklaration in diesem Beispiel besagt, dass der Datei-Inhalt der Syntax von HTML 5 folgt:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv=Content-type content="text/html; charset=us-ascii">
        <title>Meine Homepage</title>
    </head>
    <body>
        <div>
            <h1>Willkommen auf meiner <em>Homepage!</em></h1>
            Das bin <em>ich</em>:
            <br>
            <img src=photograph.png alt="Bild von mir">
        </div>
        <div>
            Und hier sind <a href=friends.html>meine Freunde.</a>
        </div>
    </body>
</html>
```

Listing 7.20: Quelltext eines HTML-Dokumentes.

Der Browser *Chrome* stellt diese Seite folgendermaßen dar:

[69] Auf `http://validator.w3.org/check` steht ein Dienst zur Verfügung, der die Korrektheit von HTML-Dateien untersucht und gegebenenfalls Auskunft zu Problemen gibt.

7.3.7 HTTP-Request

HTTP-Requests und -Responses ohne Gedächtnis

Browser und Webserver transportieren HTML-Seiten und andere Dateien und Ressourcen mit dem Protokoll HTTP.[70] Ein wesentliches Merkmal von HTTP ist seine **Zustandslosigkeit**. Das bedeutet, dass ein Webserver jeden einzelnen HTTP-Request isoliert bearbeitet, so als wäre es der erste und einzige vom betreffenden Client.[71]

Requests und Responses mit Header und Body

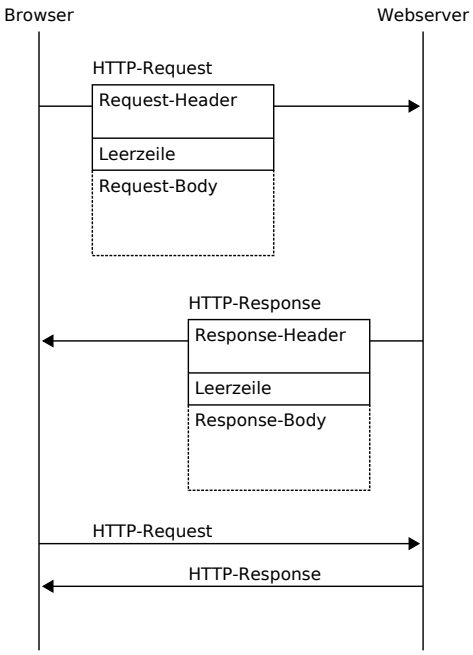
HTTP läuft als Folge von abwechselnden **Anfragen** (*request*) des Browsers und **Antworten** (*response*) des Webservers ab. Request und Response bestehen jeweils aus zwei Teilen, einem **Kopf** (*header*) und einem **Rumpf** (*body*)[72]. Die Köpfe von Request und Response sind eine Folge nicht leerer Textzeilen, gefolgt von einer Leerzeile als Trenner zwischen Kopf und Rumpf.[73] Kopf und Leerzeile sind verpflichtend, der Rumpf kann manchmal wegbleiben:

[70] Es gibt zwei HTTP-Versionen, 1.0 und 1.1, deren Unterschiede hier keine Rolle spielen.

[71] Diese Zustandslosigkeit passt keineswegs für alle Anwendungen. Um beispielsweise einen „Warenkorb“ in einem Webshop zu implementieren, muss der Server den Kunden kennen und wissen, welche Artikel er bereits in den Warenkorb gelegt hat. Cookies und andere Mechanismen umgehen die Zustandslosigkeit von HTTP.

[72] Der Begriff „body“ hat in diesem Zusammenhang nichts mit dem zufällig gleich benannten HTML-Element zu tun.

[73] In die Textnachrichten können binäre Daten eingefügt sein.



Die erste Zeile eines HTTP-Requests lautet:

```
GET /resource HTTP/1.0
```

HTTP-Verb beginnt Request-Header

Das „Verb“ `GET` wird als „HTTP-Methode“ bezeichnet.[74] Die HTTP-Methode drückt aus, welche Art von Aktion vom Webserver erwartet wird. Mit `GET` fordert ein Client eine Ressource an, die der Server als Antwort zurückschicken soll. Ein solcher `GET`-Request hat keinen Rumpf und endet mit der obligatorischen Leerzeile.

Header-Fields mit weiteren Informationen zum Request

Die weiteren Zeilen im Kopf der Anfrage werden als **Header-Fields** bezeichnet und geben zusätzliche Information über den Browser und seine Fähigkeiten. Sie sind optional. Ihr allgmeiner Aufbau ist[75]

```
name: value
```

Beispiele für Header-Fields sind:

[74] HTTP-Methoden haben nichts mit Methoden in objektorientierten Programmiersprachen zu tun. Es gibt weitere HTTP-Methoden, die hier aber nicht gebraucht werden.

[75] Groß- und Kleinschreibung in Feldnamen werden ignoriert. Im Wert nach dem Doppelpunkt kann die Schreibweise signifikant sein.

`User-Agent: El cheapo browser 1.0`
: Hersteller, Modell und Version des Browsers.

`Accept-Charset: iso-8859-1,utf-8`
: Encodings, mit denen der Browser umgehen kann.

Das folgende Beispiel zeigt einen minimalen HTTP-Request:

```
GET /index.html HTTP/1.0
(Leerzeile)
```

Ein solcher Request kann mit dem Telnet-Client direkt an eine Suchmaschine geschickt werden. Wichtig ist die Eingabe der Leerzeile nach der `GET`-Zeile:

HTTP-Request via Telnet

```
$ telnet www.google.de 80
Trying 74.125.39.147...
Connected to www.l.google.com.
Escape character is '^]'.
GET /index.html HTTP/1.0

HTTP/1.0 302 Found
Location: http://www.google.de/index.html
...

<HTML>...</HTML>
Connection closed by foreign host.
```

Der Webserver schickt eine Antwort zurück, deren Aufbau im nächsten Abschnitt erklärt wird.

7.3.8 HTTP-Response

Die Antwort eines Webservers beginnt mit dem Kopf, dessen erste Zeile lautet:[76]

HTTP-Response mit Statuscode

```
HTTP/1.0 Code Text
```

Der *Code* gibt Auskunft darüber, wie der Server mit der Anfrage zurechtkam. Die häufigsten Codes sind:[77]

- 200 (*Ok*): Request ausgeführt, der Rumpf enthält das Dokument.

[76] Die Angabe der HTTP-Version sollte sich mit der Angabe im Request decken.

[77] Die komplette Liste steht auf `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`.

- 404 (*Not found*): Das angefragte Dokument gibt es nicht.
- 500 (*Internal Server Error*): Die Anfrage war in Ordnung, aber der Server ist auf ein internes Problem gelaufen.

Codegruppen

Gruppen von ähnlichen Statuscodes

Die Codes sind nach Hunderter-Gruppen geordnet:[78]

- 200–299: Request in Ordnung, hier sind die gewünschten Informationen.
- 300–399: Request verstanden, aber so nicht erfüllbar. Bitte anders formulieren und noch einmal schicken.
- 400–499: Request unverständlich oder nicht ausführbar. Der Fehler liegt auf Clientseite, der Server kann ihn nicht beheben.
- 500–599: Request kann nicht ausgeführt werden. Das Problem liegt auf Serverseite, der Client hat alles richtig gemacht.

Header-Fields mit weiteren Informationen zum Response

Der Rest des Kopfes besteht aus einer Liste von Header-Fields, wie beim Request. Über die Reihenfolge und die konkrete Auswahl der einzelnen Header entscheidet der Server. Beispiele für häufig gelieferte Header-Fields sind:

`Content-Type: text/html; charset=UTF-8`
: Art und Encoding der zurückgelieferten Datei.

`Date: Wed, 23 Nov 2011 09:38:32 GMT`
: Zeitmarke der Antwort.

`Server: gws`
: Hersteller, Modell und Version des Webservers.

`Content-Length: 228`
: Länge des Rumpfes in Bytes.

Nach dem Kopf folgt die obligatorische Leerzeile und dann der Response-Body. Bei Codes im 200er-Bereich enthält der Rumpf die gewünschte Datei.

Browser-Prototyp kopiert Antwort auf den Bildschirm

Das folgende Programm schickt einen einfachen HTTP-Request an einen Server und nimmt die Antwort in Empfang. Der Response-Header wird auf die Standard-Fehlerausgabe geschrieben und erscheint daher immer auf dem Bildschirm. Der Response-Body wird auf die Standardausgabe kopiert und kann in eine Datei umgelenkt werden.

[78] HTTP-Version 1.1 definiert weitere Codes im Bereich 100–199, die vorläufige Antworten markieren, denen noch eine endgültige Antwort folgt.

```
import java.io.*;
import java.net.*;

public class HTTPRequest {
    public static void main(String... args) throws IOException {
        String hostname = args[0];
        String resource = args[1];
        int port = 80;
        try(Socket socket = new Socket(hostname, port);
            InputStream input = socket.getInputStream();
            BufferedReader reader = new BufferedReader(new InputStreamReader(input));
            OutputStream output = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(new OutputStreamWriter(output))) {
            // Request-Header senden
            writer.printf("GET %s HTTP/1.0%n", resource);
            writer.printf("Host: %s%n", hostname);
            writer.println("User-Agent: Java HTTPRequest");
            writer.println("Accept: text/html");
            writer.println();
            writer.flush();
            // Kein Request-Body

            // Response-Header empfangen
            for(String line = reader.readLine(); !line.isEmpty(); line = reader.readLine(
                System.err.println(line);
            // Response-Body empfangen
            for(String line = reader.readLine(); line != null; line = reader.readLine())
                System.out.println(line);
        }
    }
}
```

Listing 7.21: HTTP-Request abschicken und HTTP-Response aufnehmen.

Ein Aufruf holt die Startseite von Google und kopiert sie in eine lokale Datei. (Der folgende Abdruck ist gekürzt.)

```
$ java HTTPRequest www.google.de /index.html > google.html
HTTP/1.0 200 OK
Date: Wed, 23 Nov 2011 10:47:05 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=57fb3163ffe1feef...
Set-Cookie: NID=53=eDbYKgccnvxMR3abR...
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
$
```

Wenn Sie die Datei `google.html` in einen Browser laden, dann wird der Textinhalt der Seite lesbar dargestellt. Allerdings fehlen die Bilder und die anderen Seitenelemente, weil die referenzierten Ressourcen zwar auf dem Google-Server zu finden

sind, aber nicht in Ihrem Filesystem, wo die Datei `google.html` jetzt liegt. Die Seite ist funktionslos.

7.3.9 HTTP-Zugriff mit Bibliotheksklassen

Bibliotheksklassen für HTTP-Zugriff

Das Programm `HTTPRequest` (Listing 7.21) wickelt einen HTTP-Zugriff mit elementaren Mitteln ab. Einfacher lässt sich das mit Klassen der Java-Bibliothek erledigen. Das folgende Programm erwartet eine URL auf der Kommandozeile. Es holt die entsprechende Seite und gibt sie mithilfe der Klasse `Streams` (Listing 7.16) auf dem Bildschirm aus:

```
import java.io.*;
import java.net.*;

public class GetURL {
    public static void main(String... args) throws IOException {
        try(InputStream input = new URL(args[0]).openStream()) {
            Streams.copy(input, System.out);
        }
    }
}
```

Listing 7.22: Inhalt einer URL holen und ausgeben.

Der Ausdruck `new URL(args[0]).openStream()` im Kopf des ARM-Blocks ist eine Kurzfassung der folgenden Einzelschritte:

`URL url = new URL(args[0]);`
: Der Konstruktor verpackt die URL im String-Argument in ein `URL`-Objekt. Wenn das Argument keine zulässige URL enthält, wirft er eine `MalformedURLException`.

`URLConnection connection = url.openConnection();`
: Das `URLConnection`-Objekt repräsentiert eine Verbindung zu einem Webserver. An diesem Punkt handelt es sich nur um ein Java-Objekt. Es wird noch kein Kontakt zum Server aufgenommen.

`connection.connect();`
: Dieser Methodenaufruf stellt die Netzwerkverbindung zum Server her.

`InputStream input = connection.getInputStream();`
: Der `InputStream` liefert den Inhalt des Response-Body vom Server.

Der Rumpf der `main`-Methode in `GetURL` (Listing 7.22) entspricht also im Einzelnen dem folgenden Code:

```
URL url = new URL(args[0]);
URLConnection connection = url.openConnection();
connection.connect();
try(InputStream input = connection.getInputStream()) {
    Streams.copy(input, System.out);
}
```

Listing 7.23: Aufbau eines URL-`InputStream` in Einzelschritten.

HTTP-Request mit Zeitlimit

Das `URLConnection`-Objekt lässt sich konfigurieren, bevor `connect` aufgerufen wird. Unter anderem können die folgenden Methoden aufgerufen werden:

`void setConnectTimeout(int millis)`
: Bricht den nachfolgenden `connect`-Aufruf mit einer `SocketTimeoutException` ab, wenn der Server nicht innerhalb von `millis` Millisekunden antwortet.

`void setReadTimeout(int millis)`
: Bricht ab, wenn der Server nicht innerhalb von `millis` Millisekunden die ersten Daten liefert.

`void setRequestProperty(String name, String value)`
: Fügt ein Header-Field mit dem angegebenen Namen und Wert in den Request ein.

Nachdem die Verbindung hergestellt ist, stehen weitere Informationen zur Verfügung:

`String getHeaderField(String name)`
: Liefert den Wert des Response-Header-Fields mit dem gegebenen Namen oder `null`, wenn es kein entsprechendes Header-Field gibt.

`Map<String, List<String» getHeaderFields()`
: Liefert eine Map, die die Namen aller Response-Header-Fields auf ihre Werte abbildet. Manche Header-Fields tauchen mehrfach auf, daher stellt die Map Listen von Werten zur Verfügung. Das folgende Codefragment gibt alle Response-Header-Fields auf die Standard-Fehlerausgabe aus:

```
for(Map.Entry<String, List<String>> entry:
        connection.getHeaderFields().entrySet())
    for(String value: entry.getValue())
        System.err.printf("%s: %s%n", entry.getKey(), value);
```

Listing 7.24: Response-Header-Fields einer `URLConnection` auf die Fehlerausgabe schreiben.

`InputStream getInputStream()`
: Liefert einen `InputStream` zum Auslesen des Response-Body, wie oben bereits verwendet.

Das Programm `GetURL` (Listing 7.22) wird wie `HTTPRequest` (Listing 7.21) benutzt, wobei die erste Zeile des Response-Header wie ein „Header-Field ohne Name“ (`null`) erscheint:

```
$ java GetURL http://www.google.de > google.html
null: HTTP/1.1 200 OK
X-Frame-Options: SAMEORIGIN
Date: Fri, 25 Nov 2011 14:14:14 GMT
Transfer-Encoding: chunked
Expires: -1
X-XSS-Protection: 1; mode=block
Set-Cookie: NID=53=Vc9VQtWH_pyUO7YD_l...
Set-Cookie: PREF=ID=6e2dcdbc018e7981:...
Content-Type: text/html; charset=ISO-8859-1
Server: gws
Cache-Control: private, max-age=0
$
```

7.3.10 Minimaler Webserver

Webserver mit einer einzigen Webseite

Mit Kenntnis des HTTP lässt sich ein minimaler Webserver implementieren. Er ignoriert den Request und liefert immer die gleiche Seite zurück, die als String-Literal `responseBody` im Code gespeichert ist. Der String enthält den Quelltext von `homepage.html` (Listing 7.20).

```
import java.io.*;
import java.net.*;

public class NanoHTTPServer {
    private static final String responseBody = "<!DOCTYPE html>"
            + "<html>"
            + "    <head>"
            + "          <meta http-equiv=Content-type content=\"text/html; charset=us-as
            + "          <title>Meine Homepage</title>"
            + "    </head>"
            + "    <body>"
            + "          <div>"
            + "              <h1>Willkommen auf meiner <em>Homepage!</em></h1>"
            + "              Das bin <em>ich</em>:"
            + "              <reader>"
            + "              <img src=photograph.png alt=\"Bild von mir\">"
            + "          </div>"
            + "          <div>"
            + "              Und hier sind <a href=friends.html>meine Freunde.</a>"
            + "          </div>"
            + "    </body>"
            + "</html>";

    public static void main(String... args) throws IOException {
        int port = Integer.parseInt(args[0]);
        try(ServerSocket serverSocket = new ServerSocket(port)) {
            while(true)
                try(Socket socket = serverSocket.accept();
                    InputStream input = socket.getInputStream();
                    BufferedReader reader = new BufferedReader(new InputStreamReader(inpu
                    OutputStream output = socket.getOutputStream();
                    PrintWriter writer = new PrintWriter(new OutputStreamWriter(output)))
                    // Request-Header empfangen und ignorieren
                    for(String line = reader.readLine(); !line.isEmpty(); line = reader.r
                        ;
                    System.out.println("request from " + socket.getRemoteSocketAddress())

                    // Response-Header senden
                    writer.println("HTTP/1.0 200 OK");
                    writer.println("Content-Type: text/html; charset=ISO-8859-1");
                    writer.println("Server: NanoHTTPServer");
                    writer.println();
                    // Response-Body senden
                    writer.println(responseBody);
                }
                catch(IOException iox) {
                    // Fehler
                }
```

```
            }
        }
    }
```

Listing 7.25: Minimaler Webserver für eine einzige Seite.

Das Programm wartet nach dem Aufruf auf Requests von Clients:

```
moon$ java NanoHTTPServer 2000
```

Sobald ein Browser auf die URL

```
http://moon:2000
```

zugreift, protokolliert der Server die Zugriffe und liefert die Seite zurück.

```
request from /192.168.1.23:35195
request from /192.168.1.23:35196
```

Bei jedem Seitenaufruf im Browser registriert der Server zwei Zugriffe. Der Browser erhält zunächst die HTML-Seite und findet darin das `img`-Element. Er will daraufhin in einem zweiten getrennten Zugriff auch die Bilddatei `photograph.png` vom Server holen, erhält aber stattdessen wieder nur denselben HTML-Text, aber keine Bilddatei. Die meisten Browser sind schlau genug, um diesen Fehler zu bemerken und zu kompensieren. Sie zeigen statt des Bildes einen leeren Rahmen oder ein Symbol für eine defekte Bilddatei.

Die Portnummern zeigen die *ausgehenden* Ports des Clients, die das Betriebssystem dem Browser nach Verfügbarkeit zuweist (siehe Seite 460).

7.3.11 Statischer Webserver

Der Server `NanoHTTPServer` (Listing 7.25) funktioniert zwar im Prinzip, liefert aber immer dieselbe Seite. Ein brauchbarer Server ermittelt zunächst die im Request angeforderte Ressource und gibt dann eine entsprechende Antwort zurück.

Basisklasse mit den Grundfunktionen eines Webservers

Die folgende abstrakte Basisklasse `BaseHTTPServer` stellt die Grundfunktion eines Webservers bereit.

- Die Methode `loop` mit einer Portnummer als Parameter öffnet zunächst einen Serverport und bleibt dann endlos in einer Request-Response-Schleife. Immer wenn `accept` zurückkehrt, startet ein weiterer Zyklus. `loop` teilt die Arbeit auf die zwei folgenden Methoden auf:
- `receiveRequest` empfängt und analysiert den Request. Sie vergleicht dazu die Zeilen des Headers mit dem regulären Ausdruck `getLinePattern`, der genau zur `GET`-Zeile passt, und liefert die angeforderte Ressource als Ergebnis zurück.
- `sendResponse` schickt die Antwort zurück. Die Methode versucht, aus der Ressource mit der abstrakten Methode `getResponseBody` den Rumpf der Antwort zu berechnen. Das Ergebnis `null` bedeutet, dass die Ressource nicht verfügbar ist. `loop` liefert in diesem Fall den Response-Code 404 an den Client zurück. Ein anderes Ergebnis als `null` wird als Erfolg interpretiert und mit dem Code 200 an den Client geschickt.[79]
- Die eigentliche Arbeit steckt in der abstrakten Methode `buildResponseBody`, die die Antwortseite produziert. Abgeleitete Klassen können sie auf ihre spezifische Art implementieren.

```
import java.io.*;
import java.net.*;
import java.util.regex.*;

public abstract class BaseHTTPServer {
    public abstract byte[] buildResponseBody(String resource);

    String receiveRequest(BufferedReader reader) throws IOException {
        final Pattern getLinePattern = Pattern.compile("(?i)GET\\s+/(.*?)\\s+HTTP/1\\.[01
        String resource = null;
        for(String line = reader.readLine(); !line.isEmpty(); line = reader.readLine()) {
            Matcher matcher = getLinePattern.matcher(line);
            if(matcher.matches())
                resource = matcher.group(1);
        }
        return resource;
    }

    public void sendResponse(String resource, OutputStream output, PrintWriter writer) th
        byte[] responseBody = buildResponseBody(resource);
        if(responseBody == null) {
            writer.println("HTTP/1.0 404 Not found");
            responseBody = "Not found".getBytes();
        }
        else
```

[79] Diese Klasse ignoriert ein Problem, das sich als diffizil erweist: Abhängig vom Inhalt der tatsächlich zurückgelieferten Datei sollte das Response-Header-Field „Content-Type“ mit einem korrekten Wert gesetzt werden. Diesen Wert herauszufinden ist allerdings alles andere als einfach und erfordert einigen Aufwand. Dieser Server verdrängt das Problem: Er macht überhaupt keine Angabe zum Inhalt und verlässt sich darauf, dass der Browser schon mit der Datei zurechtkommt. In den meisten Fällen klappt das auch, weil die populären Browser sehr fehlertolerant arbeiten.

```
            writer.println("HTTP/1.0 200 OK");
        writer.println("Server: " + getClass().getSimpleName());
        writer.println("Content-Length: " + responseBody.length);
        writer.println();
        writer.flush();
        output.write(responseBody);
    }

    public void loop(int port) throws IOException {
        try(ServerSocket serverSocket = new ServerSocket(port)) {
            while(true)
                try(Socket socket = serverSocket.accept();
                    InputStream input = socket.getInputStream();
                    BufferedReader reader = new BufferedReader(new InputStreamReader(inpu
                    OutputStream output = socket.getOutputStream();
                    PrintWriter writer = new PrintWriter(new OutputStreamWriter(output)))
                    sendResponse(receiveRequest(reader), output, writer);
                }
        }
    }
}
```

Listing 7.26: ABC mit der Grundfunktionalität eines elementaren Webservers.

Webserver für Dateien aus dem Filesystem

Als konkretes Beispiel leitet die Klasse `FileHTTPServer` den `BaseHTTPServer` ab. `FileHTTPServer` implementiert einen statischen Webserver, der Ressource-Strings auf das Server-Filesystem bezieht und den Inhalt von Dateien mit den entsprechenden Pfadnamen zurückliefert.

```
import java.io.*;
import java.nio.file.*;

public class FileHTTPServer extends BaseHTTPServer {
    public byte[] buildResponseBody(String resource) {
        try {
            return Files.readAllBytes(Paths.get(resource));
        }
        catch(IOException e) {
            return null;
        }
    }

    public static void main(String... args) throws IOException {
        new FileHTTPServer().loop(Integer.parseInt(args[0]));
    }
}
```

Listing 7.27: Webserver für statische Seiten aus dem Server-Filesystem.

Wenn man `FileHTTPServer` zum Beispiel in dem Directory startet, in dem `homepage.html` (Listing 7.20) abgespeichert ist, dann stellt er diese Seite allen Browsern unter der URL

```
http://moon:2000/homepage.html
```

zur Verfügung. Auch das Bild ist korrekt eingebettet, wenn die Bilddatei `photograph.png` im gleichen Directory liegt.

Document-Root zur Einschränkung

Starten Sie diesen Server nur in einem geschützten Netzwerk und keinesfalls auf einem öffentlich erreichbaren Rechner! Er wird ohne mit der Wimper zu zucken alle Dateien von Ihrer Festplatte ausliefern, wenn er sie nur lesen kann.

Eine erste Schutzmaßnahme wäre ein fest vorgegebenes Directory (*document root*), auf das sich alle Ressource-Pfade beziehen. Darüber hinaus sollten die ausgelieferten Dateien eine maximale Länge nicht überschreiten. Schließlich könnten einmal ausgelieferte Dateien in einen Cache kopiert werden (siehe Seite 293), um das verhältnismäßig langsame Lesen von Dateien einzusparen.

7.3.12 Dynamische Webseiten

Seiten mit berechnetem Inhalt

Die bisher entwickelten Webserver liefern zu jeder Ressource immer die gleiche Antwort. Sie heißen deshalb statische Webserver. Im Gegensatz dazu *berechnen* dynamische Webserver die Antwort bei jedem Request. Das kann bedeuten, dass eine Ressource für einen Client bei jedem Zugriff anders aussieht.[80]

Ein einfaches Beispiel ist ein „Uhrzeit-Server“, der eine HTML-Seite zurückliefert, auf der die aktuelle Serverzeit angegeben ist.

```
import java.io.*;
import java.util.*;

public class ClocktimeHTTPServer extends BaseHTTPServer {
    private static final String pageFormat = "<!DOCTYPE html>"
            + "<html>"
            + "    <head>"
            + "        <meta http-equiv=Content-type content=\"text/html; charset=us-asci
            + "        <title>Clocktime-Server</title>"
            + "    </head>"
            + "    <body>"
            + "        <div>"
            + "            <h1>Clocktime-Server</h1>"
            + "            Die aktuelle Server-Zeit ist <strong>%s</strong>"
            + "        </div>"
            + "    </body>"
            + "</html>%n";

    public byte[] buildResponseBody(String resource) {
```

[80] Das bedeutet auch, dass ein Client solche Seiten nicht unbedingt zwischenspeichern kann und bei jedem Zugriff aufs Neue vom Server anfordern muss.

```
        return String.format(pageFormat, new Date()).getBytes();
    }

    public static void main(String... args) throws IOException {
        new ClocktimeHTTPServer().loop(Integer.parseInt(args[0]));
    }
}
```

Listing 7.28: Dynamischer Webserver, der die aktuelle Serverzeit liefert.

Dieser Server liefert bei jedem Zugriff eine neue Seite. Das lässt sich mit dem *Reload*-Button des Browsers überprüfen.

Webserver für „aufwendige“ Berechnungen

Als weiteres Beispiel eines dynamischen Webservers soll ein Fibonaccizahlen-Server implementiert werden, der Zahlen mit der Fibonacciformel berechnet und liefert.[81] Als Ressource akzeptiert der Server Pfadangaben der Art

```
/a/b/n
```

Dabei sind a und b die Startwerte einer Zahlenfolge, deren n-tes Element angefordert wird. Beispielsweise steht unter dem Pfad

```
/1/3/6
```

die sechste Zahl der Lucas-Folge, die den Wert 18 hat, zur Verfügung (siehe Seite 281). Auch dieser Server nutzt die Basisklasse `BaseHTTPServer` (Listing 7.26). Die Fibonaccizahlen selbst berechnet die Klasse `AnyFibonacci` (Listing 4.50):

```
import java.io.*;

public class FibonacciHTTPServer extends BaseHTTPServer {
    private static final String pageFormat = "<!DOCTYPE html>"
            + "<html>"
            + "     <head>"
            + "         <meta http-equiv=Content-type content=\"text/html; charset=us-asc
            + "         <title>Fibonaccizahlen-Server</title>"
            + "     </head>"
            + "     <body>"
            + "         <div>"
            + "             <h1>Fibonaccizahlen-Server</h1>"
            + "             Die %d. Zahl der Fibonaccifolge mit den Startwerten %d, %d is
            + "             <br>"
            + "             <strong>%d</strong>"
            + "         </div>"
```

[81] Die Berechnung der Fibonaccizahlen steht hier stellvertretend für eine Aufgabe, die spürbar Rechenaufwand erfordert und eine Weile dauert. Ein gut ausgestatteter Server bewältigt solche Aufgaben leichter als ein vielleicht leistungsschwacher Client.

```
            + "     </body>"
            + "</html>%n";

    public byte[] buildResponseBody(String resource) {
        try {
            String[] token = resource.split("/");
            int a = Integer.parseInt(token[0]);
            int b = Integer.parseInt(token[1]);
            int n = Integer.parseInt(token[2]);
            long result = new AnyFibonacci(a, b).fib(n);
            return String.format(pageFormat, n, a, b, result).getBytes();
        }
        catch(Exception ex) {
            return null;
        }
    }

    public static void main(String... args) throws IOException {
        new FibonacciHTTPServer().loop(Integer.parseInt(args[0]));
    }
}
```

Listing 7.29: Fibonaccizahlen-Webserver.

Nach dem Start auf dem Server `moon` liefert der Zugriff auf `http://moon:2000/1/3/6` eine freundliche Seite mit der gesuchten Lucas-Zahl 18.

7.3.13 Nebenläufiger Webserver

Zeitaufwendige Seite blockiert andere Besucher

Die Seite `http://moon:2000/1/3/6` des Servers `FibonacciHTTPServer` (Listing 7.29) wird schnell geliefert. Das Laden von `http://moon:2000/1/1/50` dauert dagegen deutlich länger. Greift man in dieser Zeit mit einem anderen Browser zur gleichen Zeit auf die „schnelle“ Seite `http://moon:2000/1/3/6` zu, dann erhält man zunächst keine Antwort, weil der `FibonacciHTTPServer` mit der Berechnung der Antwort des vorhergehenden Requests beschäftigt ist und sich in dieser Zeit um keine Anfragen kümmert. Das bedeutet, dass ein Besucher, der eine aufwendige Seite abruft, alle anderen Besucher damit ausschließt.

Neuer Thread für jeden Request

Dieses Problem lässt sich mit Threads lösen. Statt alle Requests *nacheinander* zu verarbeiten, startet ein nebenläufiger Server für jeden Request einen neuen Thread. Im Wesentlichen wird dazu der Schleifenrumpf der `loop`-Methode der Klasse `BaseHTTPServer` (Listing 7.26) zur `run`-Methode eines Threads. Das `Thread`-Objekt selbst ist nicht weiter interessant, deshalb reicht eine anonyme Klasse aus. Die folgende Codeskizze zeigt die Idee:

```
public void loop(int port) throws IOException {
    try(ServerSocket ss = new ServerSocket(port)) {
```

```
            while(true)
                new Thread() {
                    public void run() {
                        // ehemaliger Schleifenrumpf
                    }
                }.start();
        }
    }
```

Dabei sind noch ein paar technische Probleme zu lösen:

- Die `run`-Methode darf keine Checked-Exception werfen. `run` fängt deshalb eine `IOException` selbst auf und verpackt sie in eine `RuntimeException`.[82]
- Der `accept`-Aufruf in `run` käme zu spät. Er muss aus `run` heraus und *vor den Start* des Threads verschoben werden. Andernfalls würde die Schleife ungebremst Threads starten, von denen jeder erst *nach dem Start* auf einen `accept`-Aufruf läuft und dann wartet. Das Laufzeitsystem würde mit Threads überschwemmt und schnell zusammenbrechen.
- Obwohl der von `accept` gelieferte Socket außerhalb des Threads definiert ist, muss ihn der Thread am Ende der `run`-Methode selbst schließen. Das ARM lässt sich damit leider nicht mehr anwenden, weil Öffnen und Schließen des Sockets nicht im gleichen Block liegen können.

Schnellere Reaktion eines nebenläufigen Servers

Insgesamt ergibt sich die folgende Implementierung der Basisklasse eines Webservers mit mehreren Threads:

```
import java.io.*;
import java.net.*;

public abstract class BaseParallelHTTPServer extends BaseHTTPServer {
    public abstract byte[] buildResponseBody(String resource);

    public void loop(int port) throws IOException {
        try(ServerSocket serverSocket = new ServerSocket(port)) {
            while(true) {
                final Socket socket = serverSocket.accept();
                new Thread() {
                    public void run() {
                        try(InputStream input = socket.getInputStream();
                            BufferedReader reader = new BufferedReader(new InputStreamRea
                            OutputStream output = socket.getOutputStream();
                            PrintWriter writer = new PrintWriter(new OutputStreamWriter(o
                            sendResponse(receiveRequest(reader), output, writer);
                            socket.close();
```

[82] Das ist im Allgemeinen keine allzu gute Idee, weil `IOException` und `RuntimeException` ganz unterschiedliche Arten von Schwierigkeiten signalisieren. In diesem Fall bleibt aber wegen der Signatur von `run` keine andere Wahl.

```
                        }
                        catch(IOException iox) {
                            throw new RuntimeException(iox);
                        }
                    }
                } .start();
            }
        }
    }
}
```

Listing 7.30: ABC mit der Grundfunktionalität eines elementaren nebenläufigen Webservers.

Für die abgeleiteten Klassen ändert sich nichts. Ein parallel arbeitender Fibonaccizahlen-Server liefert `http://moon:2000/1/3/6` sofort, auch während ein anderer Besucher noch auf `http://moon:2000/1/1/50` wartet.[83]

7.3.14 Gefährlicher dynamischer Server

Betriebssystem-Kommandos als Ressourcen

Ressourcen können beliebige Strings sein. Ihre Interpretation ist alleine Sache des Webservers. Der folgende Server interpretiert den Ressource-String als Kommando des Betriebssystems. Er führt das Kommando aus und liefert die Ausgabe des Kommandos in einer Antwortseite an den Client zurück.

```
import java.io.*;
import java.net.*;

public class CommandHTTPServer extends BaseParallelHTTPServer {
    public byte[] buildResponseBody(String resource) {
        System.out.println(resource);
        try {
            String command = URLDecoder.decode(resource, "utf-8");
            Process process = new ProcessBuilder(command.split("\\s+"))
            .redirectErrorStream(true)
            .start();
            process.waitFor();
            ByteArrayOutputStream output = new ByteArrayOutputStream();
            Streams.copy(process.getInputStream(), output);
            return output.toByteArray();
        }
        catch(IOException | InterruptedException e) {
            return null;
        }
    }

    public static void main(String... args) throws IOException {
        new CommandHTTPServer().loop(Integer.parseInt(args[0]));
```

[83] Erst wenn alle Prozessoren ausgelastet sind, bremst der Server ab.

```
    }
}
```

Listing 7.31: Webserver, der Shell-Kommandos ausführt.

Übermittlung von Sonderzeichen in URLs

Dabei ist zu beachten, dass Leerstellen und andere Sonderzeichen in URLs durch Zeichensequenzen der Art

```
%xx
```

mit hexadezimalen Codes *xx* ersetzt werden. Leerzeichen erscheinen beispielsweise als `%20` in URLs. Die statische Methode `URLDecoder.decode` der Java-Bibliothek decodiert diese Ersatzschreibweisen und liefert den Originalstring.[84]

Zum Beispiel liefert der Zugriff auf

```
http://moon:2000/jps -m
```

auf dem Linux-System des Autors die folgende Antwort:

```
14436 Jps -m
13886 CommandHTTPServer 2000
31279 Main --userdir /home/hidden/.netbeans/7.0 --branding nb
```

Dieser Server ist gefährlich, weil er die Fernsteuerung der Servers über jeden Browser zulässt. Der Zugriff auf

```
http://moon:2000/jkill CommandHTTPServer
```

stoppt den Server selbst. Ein Reload der Seite liefert dann einen Zugriffsfehler. Je nach Konfiguration des Systems hält

```
http://moon:2000/halt
```

das ganze Serversystem an! Mit angemessener Absicherung liegt in dieser Freiheit aber auch der Nutzen dieses Webservers.

Anders als bei dem weiter oben entwickelten `RemoteCommandServer` (Listing 7.17) reicht hier jeder Browser als Client aus. Ein spezieller Client wie `RemoteCommandClient` (Listing 7.15) ist nicht nötig.

[84] Entsprechend fügt `URLEncoder.encode` URL-Escapecodes ein.

7.3.15 Webserver-Basisklasse

Webserver als Basisklasse in der Bibliothek

Mit Kenntnis von HTTP lassen sich viele nicht triviale Programme mit einem „Web-Frontend“ ausstatten, das programminterne Informationen auf portable Art über ein Netzwerk zur Verfügung stellt. Allerdings ist die Implementierung von HTTP auf der Grundlage von Sockets zwar flexibel, aber etwas mühsam. Leichter fällt das mit der Klasse `HttpServer` aus der Java-Laufzeitbibliothek, die einen kompletten Webserver mit einigen interessanten Eigenschaften zur Verfügung stellt.[85]

Nicht-Standard-Package

Allerdings findet sich diese Klasse im Package `com.sun.net.httpserver` und nicht in der Package-Hierarchie unter einem der Toplevel-Pakete `java` und `javax`. Das bedeutet, dass diese Klassen in Java-Implementierungen anderer Hersteller als Oracle möglicherweise fehlen. Unter diesem Vorbehalt erweist sich die Klasse `HttpServer` als ausgesprochen nützlich.

Ein `HttpServer`-Objekt setzt das Protokoll HTTP um. Den eigentlichen Inhalt der Antworten produzieren „Handler“-Objekte, die der Entwickler definiert und vor dem Start des Servers an bestimmte Ressource-Strings „bindet“. Der `HttpServer` arbeitet gewissermaßen als Verteiler. Er nimmt einen Request an, untersucht den Ressource-String, sucht den zuständigen Handler heraus und ruft ihn auf. Der Handler produziert den Inhalt, den der Server in eine HTTP-Antwort verpackt und zum Client zurückschickt.

Die folgende Klasse implementiert einen Uhrzeit-Webserver mit der gleichen Funktionalität wie `ClocktimeHTTPServer` (Listing 7.28). (Die Klasse `DateHandler` wird weiter unten beschrieben.)

```
import com.sun.net.httpserver.*;
import java.io.*;
import java.net.*;
import java.util.*;

public class ClocktimeSunServer {

    public static void main(String... args) throws IOException {
        int port = Integer.parseInt(args[0]);
        InetSocketAddress address = new InetSocketAddress(port);
        HttpServer server = HttpServer.create(address, 0);
        server.createContext("/", new DateHandler());
        server.start();
    }
}
```

Listing 7.32: Uhrzeit-Webserver auf der Grundlage des Sun-`HttpServer`.

Die `main`-Methode geht in folgenden Schritten vor:

[85] Beispielsweise beherrscht dieser Webserver außer dem Protokoll `http` auch die abgesicherte Variante `https`.

- Ein `InetSocketAddress`-Objekt repräsentiert einen Serverport (siehe auch Seite 467).
- Die statische Factory-Methode `create` liefert ein `HttpServer`-Objekt für diesen Port. Das zweite Argument legt fest, wie viele noch nicht verarbeitete Anfragen der Server aufstauen kann, bevor er weitere Anfragen wegen Überlastung abweist. Der Wert 0 übernimmt eine Voreinstellung des Betriebssystems.
- `createContext` verknüpft Ressource-Strings mit Handlern. In diesem Beispiel gibt es nur einen Handler, der für alle Ressourcen zuständig ist.
  `createContext` kann mehrmals aufgerufen werden. Bei konkurrierenden Einträgen gilt der, bei dem das längste Präfix mit dem tatsächlichen Ressource-String übereinstimmt.
- Schließlich startet `start` den Server.

Handler berechnen Ressourcen

Der Löwenanteil der Arbeit steckt in den Handlern. Alle Handlerobjekte implementieren das Interface `HttpHandler` mit der einzigen Methode:

```
void  handle(HttpExchange exchange) throws IOException
```

Der Parameter vom Typ `HttpExchange` repräsentiert Request und Response der Client-Verbindung. `HttpExchange` definiert unter anderem die folgenden Methoden:

`void sendResponseHeaders(int code, long length)`
: Schließt den Response-Header mit dem Code `code` ab. Das zweite Argument legt die Länge des nachfolgenden Response-Body als Anzahl Bytes fest. Die Angabe 0 steht für den ganzen Stream, siehe nächster Parameter.

`OutputStream getResponseBody()`
: Liefert einen Stream, der zum Response-Body führt. `handle` schreibt die Antwort auf diesen Stream. Er muss geschlossen werden!

Der `ClocktimeSunServer` braucht nur eine einzige Handlerklasse. Diese Klasse `DateHandler` ist zur Vereinfachung als geschachtelte Klasse definiert:

```
private static class DateHandler implements HttpHandler {
    private static final String pageFormat = // wie in ClocktimeHTTPServer ...

    public void handle(HttpExchange exchange) throws IOException {
        exchange.sendResponseHeaders(200, 0);
        try(OutputStream output = exchange.getResponseBody()) {
            output.write(String.format(pageFormat, new Date()).getBytes());
        }
```

```
    }
}
```

Listing 7.33: Statisch geschachtelte Handlerklasse.

Sie ist minimal gehalten:

- Der Aufruf von `sendResponseHeaders` schließt einen Header mit Code 200 (OK) und der Default-Länge ab. Der Header enthält keine weiteren Header-Fields.
- Die Byte-Darstellung der Antwortseite nimmt der `OutputStream` auf, den `getResponseBody` liefert.

Zusammenfassung

- Das etablierte **ISO/OSI-Schichtenmodell** definiert einen **Protokollstapel,** der Netzwerkkommunikation in verschiedene Ebenen gliedert.
- Rechner in einem IP-Netzwerk identifizieren Netzwerkschnittstellen mit **Hardware- und IP-Adressen**.
- Das **Loopback-Device** simuliert eine Netzwerkverbindung ohne Hardware und dient zu Testzwecken.
- **DNS-Server** verknüpfen IP-Adressen mit **symbolischen Hostnamen**.
- Der Datenfluss in einem TCP-Netzwerk ist auf **Ports** aufgeteilt, denen **Protokolle** und **Dienste** zugeordnet sind.
- ***Well-known* Ports** sind für populäre Dienste reserviert. Das Angebot von Diensten auf ***priviledged* Ports** erfordert Administratorrechte.
- In **Client-Server-Systemen** wartet ein passiver Server auf **Requests** von aktiven Clients und antwortet darauf mit **Responses**.
- Ein `Socket`**-Objekt** repräsentiert ein Ende einer Netzwerkverbindung.
- Ein `Socket` stellt einen `InputStream` und einen `OutputStream` zum **Datentransport** bereit.
- Bei Socket-Streams sind `flush` und `close` sehr wichtig.
- Ein **Telnet-Client** ist ein nützliches Werkzeug zum interaktiven Test von Servern.
- Die Methode `accept` der Klasse `ServerSocket` wartet auf eingehende Requests und **spaltet** dann einen normalen **Socket ab,** über den die eigentliche Kommunikation läuft.
- **Cipherstreams verschlüsseln** die Daten beim Transport über das Netzwerk.
- **Webserver** liefern Ressourcen, die mit **URLs** verknüpft sind.
- Webseiten sind Textdateien in **HTML-Syntax,** die ein **Webbrowser** empfängt und darstellt.

- Webserver und Webbrowser kommunizieren mit **HTTP**.
- Die **Klasse** `URL` implementiert HTTP auf Clientseite.
- Ein Programm, das Daten mit einem **Web-Frontend** via HTTP bereitstellt, kann mit beliebigen Browsern benutzt werden.
- Ein **dynamischer Webserver** berechnet die Antwort aus der URL bei jedem Abruf neu.
- **Nebenläufige Webserver** kommen auch mit parallelen Anfragen gut zurecht.
- Die **Basisklasse** `HTTPServer` in der Bibliothek des Oracle JDK implementiert einen leistungsfähigen **Webserver**.

Aufgaben

Aufgabe 1: Datei über das Netzwerk kopieren

Gelegentlich soll eine Datei über das Netzwerk zu einem anderen Rechner geschickt werden. Dafür gibt es eine Reihe von Diensten, Protokollen und Anwendungen (ftp, scp, netcat), die aber nicht immer und überall zur Verfügung stehen. Entwickeln Sie ein Java-Programm `NetCopy`, das eine Datei von einem Rechner auf einen anderen kopiert. Beginnen Sie mit dem folgenden Programmrahmen:

```
import java.io.*;

public class NetCopy {
    public static void main(String... args) {
        String filename = null;
        String hostname = null;
        final int port = 2910;
        for(String arg: args)
            if(arg.startsWith("@"))
                hostname = arg.substring(1);
            else
                filename = arg;

        InputStream source;
        OutputStream destination;
        if(hostname == null) {
            // Datei empfangen ...
        }
        else {
            // Datei senden ...
        }
    }
}
```

Listing 7.34: Programmrahmen für ein Netzwerk-Kopierprogramm.

Dieses Programm erwartet ein oder zwei Kommandozeilenargumente und arbeitet abhängig von der Anzahl als Sender oder als Empfänger. Beim Aufruf mit einem einzigen Kommandozeilenargument, einem Dateinamen, wartet es als Empfänger auf Daten und speichert diese in der angegebenen Datei ab:

```
receiver$ java NetCopy filename
```

Beim Aufruf mit zwei Kommandozeilenargumenten, einem Hostnamen und einem Dateinamen, schickt es als Sender die angegebene Datei zum entsprechenden Rechner (`receiver`). Zur Kennzeichnung wird dem Hostnamen ein `@`-Zeichen vorangestellt:

```
sender$ java NetCopy @receiver filename
```

Nutzen Sie die Hilfsklasse `Streams` (Listing 7.16) zum Kopieren der Daten von der Datei auf die Netzwerkverbindung beziehungsweise umgekehrt.

Arbeitsweise als Filter

Erweitern Sie das Programm, sodass es auch ohne Dateinamen als Kommandozeilenargumente auskommt. In diesem Fall liest `NetCopy` Daten von der Standardeingabe (`System.in`) und sendet sie auf das Netzwerk beziehungsweise empfängt Daten vom Netzwerk und schreibt sie auf die Standardausgabe (`System.out`). Mit dieser Erweiterung kann `NetCopy` auf einem Vermittler-Rechner als *Forwarder* arbeiten, der Daten von einem Sender zu einem Empfänger weiterreicht. Das ist sinnvoll, wenn zwei Rechner in unterschiedlichen Netzwerken stehen und sich gegenseitig nicht direkt erreichen können, aber ein dritter Rechner (`forwarder`) Verbindungen zu beiden Netzwerken hat. Der Sender schickt eine Datei zum Vermittler:

```
sender$ java NetCopy @forwarder < filename
```

Der Forwarder kopiert die Datei in einer Pipe (siehe Anhang B) weiter:

```
forwarder$ java NetCopy | java NetCopy @receiver
```

Der Empfänger speichert die Daten in einer lokalen Datei:

```
receiver$ java NetCopy > filename
```

Viele Archivierungsprogramme können Archive auf die Standardausgabe ausgeben beziehungsweise von der Standardeingabe lesen. Ein Beispiel ist das `tar`-Programm von Unix oder `p7zip`, das für viele Systeme zur Verfügung steht. Damit können komplette Directorybäume über das Netzwerk übertragen werden:

```
sender$ tar cf - subdir | java NetCopy @receiver
```

```
receiver$ java NetCopy | tar xvf -
```

Sie müssen also nichts weiter implementieren, wenn beim Sender und beim Empfänger das gleiche Archivierungsprogramm zur Verfügung steht.

Kompression

Manche Daten können wirksam komprimiert werden, bei anderen klappt das weniger gut.[86] Oft fällt der Aufwand zur Kompression und Expansion gegenüber dem Aufwand zur Netzwerkübertragung nicht ins Gewicht. Erweitern Sie `NetCopy` um einen optionalen Schalter `-c`, der auf Senderseite Kompression und auf Empfängerseite Expansion einschaltet. Nutzen Sie dazu `GZIPStreams` (siehe Abschnitt 1.4.4). Natürlich müssen sich Sender und Empfänger einig sein, ob die Daten komprimiert übertragen werden oder nicht. Die Hilfe eines Dienstprogramms, wie `tar` oder `p7zip`, ist dazu nicht nötig.

Flexibler Aufruf

In der bisher entwickelten Form muss der Empfänger (Server) bereits laufen, wenn der Sender (Client) startet. Das ist nicht sehr flexibel. Erweitern Sie `NetCopy` so, dass

- der Sender eine Minute lang in regelmäßigen Abständen versucht, den Empfänger zu erreichen, wenn das nicht sofort gelingt. Nach einer Minute gibt er aber auf und bricht mit einer Fehlermeldung ab. Fangen Sie dazu die Exception auf, die der `Socket`-Konstruktor wirft, wenn der Verbindungsaufbau scheitert.

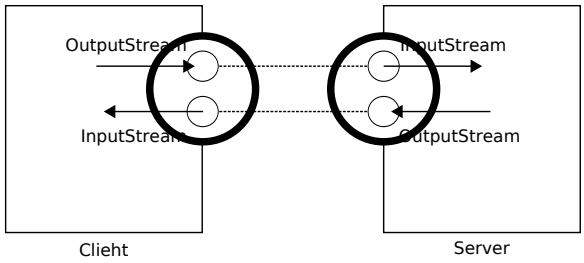
[86] Kaum komprimieren lassen sich viele Mediendaten, wie MP3-Musik, AVI-Filme oder JPEG-Bilder. Auch Zip- und Jar-Dateien sind in der Regel komprimiert. Weiter eignen sich verschlüsselte Daten nicht, wie beispielsweise Truecrypt-Volumes, weil die Verschlüsselung Regelmäßigkeiten einebnet und daher kein Ansatzpunkt für Kompression mehr bleibt.

- der Empfänger nach einer Minute aufgibt, wenn bis dahin keine Daten eingegangen sind. Verwenden Sie dazu die Methode `setSoTimeout` der Klasse `ServerSocket`.

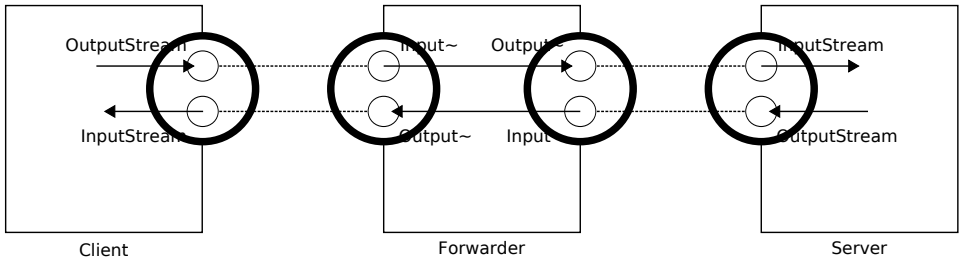
Mit dieser Erweiterung können Sender und Empfänger in beliebiger Reihenfolge starten, solange das in einem Zeitfenster von einer Minute geschieht.

Aufgabe 2: Forwarder

In einer normalen Client-Server-Verbindung kapseln die Sockets zwei gegenläufige Streams:



Mit einem dazwischengeschalteten Rechner, der die Daten in beiden Richtungen unverändert durchreicht, kann die Kommunikation überwacht werden. Ein solches System bezeichnet man als „Forwarder“. Er verhält sich aus Sicht des Clients wie der Server und aus der Sicht des Servers wie ein Client.



Ein Forwarder kann verschiedene Aufgaben übernehmen, wie zum Beispiel

- Client und Server in unterschiedlichen Netzwerken verbinden, wenn er auf einem Rechner läuft, der beide „sieht“,
- den Datenaustausch zwischen Client und Server überwachen oder beeinflussen,

- die Dienste des Servers gegenüber dem Client auf einem anderen als dem Originalport verfügbar machen.

Entwickeln Sie einen universellen Forwarder, der den durchlaufenden Datenverkehr auf der Standardausgabe mitprotokolliert. Der Forwarder ist Server und Client in einem. Er erwartet beim Start drei Kommandozeilenargumente:

1. den Port, auf dem er selbst in seiner Rolle als Server Client-Verbindungen annimmt,
2. den Hostnamen des „echten“ Servers, mit der sich in seiner Rolle als Client verbindet und
3. den Port des echten Servers.

Beachten Sie dabei,

- dass der Forwarder das Protokoll zwischen Client und Server nicht kennt. Er weiß insbesondere nicht, wann der Client Daten und wann der Server Daten schickt. Das heißt, dass er zwei Threads (siehe Kapitel 6) braucht, von denen jeder Daten in eine Richtung (Client zum Server und Server zum Client) transportiert.
- dass Client und Server die beiden Streams eines Sockets nicht unbedingt gleichzeitig schließen. Wenn einer der beiden Inputstreams endet, muss der Forwarder auch den weiterführenden `OutputStream` beenden. `close` ist dazu ungeeignet, wie in Abschnitt 7.2.11 deutlich wurde.

Der Forwarder schreibt die durchkopierten Daten auf die Standardausgabe. Allerdings weiß er nicht, ob es sich dabei um Text- oder Binärdaten handelt. Im einfachsten Fall geben Sie eine Liste der Bytewerte aus. Eine bessere Version des Forwarders untersucht zuerst die Daten und überprüft, ob es sich dabei höchstwahrscheinlich um Text handelt oder eher nicht. Die Bytewerte in ASCII-codiertem Text liegen beispielsweise alle zwischen 32 und 126, zuzüglich einiger weniger Kontrollzeichen (Newline, Return, Tabulator). Wenn nur diese Bytewerte vorkommen, geht es bestimmt um Text.

Test

Ein Forwarder zeigt deutlich, dass die Kommunikation zwischen einem `RemoteCommandClient` (Listing 7.15) und einem `RemoteCommandServer` (Listing 7.17) im Klartext abläuft:

```
moon$ java RemoteCommandServer 4000
```

Auf einem zwischengeschalteten Rechner namens `orbit` läuft der Forwarder:

```
orbit$ java Forwarder 4000 moon 4000
```

Der Clientrechner wendet sich an den Forwarder:

```
earth$ java RemoteCommandClient orbit 4000 ls
ch01basics
ch02sockets
ch03http
```

Der Forwarder zeigt einen Mitschnitt des Datenaustauschs. Die Markierungen `->` und `<-` fügt er zur besseren Übersicht ein:

```
orbit$ java Forwarder 4000 moon 4000
->ls
<-ch01basics
ch02sockets
ch03http
```

Bei den Programmen `RemoteAESPasswordCommandClient` (Listing 7.19) und `RemoteAESPasswordCommandServe` (Listing 7.18) fließen nur noch unlesbare Binärdaten:

```
orbit$ java Forwarder 4000 moon 4000
->B6 2A E0 C7 F1 3B 23 B1
->6F 5D ED 68 F8 FC AE 56 6C 2C 59 8E F8 51 D5 DD
E5 88 60 D9 D8 E3 73 6F
<-B0 4C C3 D0 23 EC AB 15
<-19 C7 42 0F D6 38 B2 AD 42 F7 44 47 FB 64 E7 F0
C5 8F A4 24 35 51 AF 82 05 0D 31 E9 AD C9 7D 6B
05 02 55 EE 77 28 F9 94 0B B0 7B 7B A4 9B 11 F9
04 CA 17 EB 59 D4 C7 98 BD F4 A6 6F E6 38 E8 6D
F8 35 D2 70 B5 09 32 8F
```

Für diesen Test sind nicht unbedingt drei verschiedene Rechner nötig. Auch drei Eingabeaufforderungen auf dem gleichen System reichen aus, um die Kommunikation zu überprüfen. Als Hostnamen verwenden Sie `localhost` oder eines der Synonyme. Allerdings müssen der „echte“ Server und der Forwarder jetzt auf unterschiedlichen Ports lauschen:

```
$ java RemoteCommandServer 4000
```

In einer anderen Eingabeaufforderung läuft der Forwarder:

```
$ java Forwarder 3999 localhost 4000
```

Der Client startet in einer dritten Eingabeaufforderung:

```
$ java RemoteCommandClient localhost 3999 ls
ch01basics
ch02sockets
ch03http
```

Die Ausgaben sind die gleichen wie vorher.

Aufgabe 3: Farbbild-Webserver

Dynamische Webseiten berechnet der Server bei jedem Zugriff neu auf der Grundlage der URL und anderer Informationen. Entwickeln Sie ausgehend vom `BaseHTTPServer` (Listing 7.26) einen Server, der URLs der Form

```
/red/green/blue/size/format
```

akzeptiert, wobei *red*, *green* und *blue* Bytewerte (0 bis 255) sind. Der Server stellt unter jeder URL eine quadratische Bilddatei einer Kantenlänge von *size* Pixel im Format *format* zur Verfügung. Als Bildformate sind `jpg`, `png` und `gif` zulässig, weil die Bibliotheksmethode `ImageIO.write` in der Voreinstellung mit diesen zurechtkommt. [87] Auch kürzere URLs sind erlaubt, wobei die fehlenden Komponenten durch Voreinstellungen ersetzt werden (*red* = *green* = *blue* = 0, *size* = 256, *format* = `png`).

Das Bild ist komplett mit der Farbe gefüllt, deren Primärfarbenkomponenten die URL vorgibt. Beispielsweise findet man unter

```
http://localhost:2000/160/200/255/500/jpg
```

ein JPEG-Bild mit 500x500 Pixel Größe. Das Bild ist blassblau, weil Rot = 160, Grün = 200 und Blau = 255 die Farbanteile von Blassblau sind. Auf

[87] Weitere Bildformate können installiert werden. Verpflichtend sind nur die genannten drei Formate.

```
http://localhost:2000/255/224
```

steht ein goldfarbenes `png`-Bild zur Verfügung.

Das folgende Codefragment schreibt das Binärabbild einer einfarbigen Bilddatei in den Bytestrom `bytesOuput`:

```
int rgb = ((255 << 8 | red) << 8 | green) << 8 | blue;
BufferedImage image = new BufferedImage(size, size, BufferedImage.TYPE_INT_ARGB);
for(int y = 0; y < size; y++)
    for(int x = 0; x < size; x++)
        image.setRGB(x, y, rgb);
ByteArrayOutputStream bytesOutput = new ByteArrayOutputStream();
ImageIO.write(image, format, bytesOutput);
```

Listing 7.35: Generieren einer Bilddatei mit der Farbe (*red*, *green*, *blue*).

Der Server hält Abermillionen verschiedener Bilddateien bereit. Versuchen Sie nicht, ihn zu „spiegeln“.

Kapitel

8 Reflection

Lernziele

In diesem Kapitel lernen Sie

- wie ein Java-Programm mit **Reflection** Objekte erzeugen kann, deren Typen nicht im Quelltext stehen, sondern erst zur Laufzeit festgelegt werden.
- welche Methoden Java bietet, um den Aufbau einer **unbekannten Klasse zu analysieren** und sie in ihre Bestandteile zu zerlegen.
- wie Sie mit Reflection die Interna beliebiger **Objekte untersuchen und verändern** können.

Unter den Begriff „Reflection“ fallen Sprachmittel, mit denen ein Java-Programm die statische Struktur von Klassen analysieren und Objekte sowohl analysieren als auch modifizieren kann. Das bedeutet zum Beispiel, dass ein Programm eine neue Klasse laden, ihren Aufbau erforschen, Objekte erschaffen und deren Methoden aufrufen kann. Den Schlüssel dazu liefert der Java-Bytecode, der weit mehr Informationen enthält[1] als beispielsweise die ausführbare Binärdatei, die ein typischer C-Compiler produziert. Java-Reflection bietet zwei Ansätze, die Anwendung auf Code und die Anwendung auf Objekte.

Die Introspektion einer übersetzten Klasse oder eines Interface bildet die Codestruktur in Objekte ab. Die Typen dieser Objekte spiegeln die Bausteine einer Java-Klasse wider, wie beispielsweise `Class`, `Method` und `Field` (Objekt- oder Klassenvariable). Sie sind überwiegend im Package `java.lang.reflect` definiert. Diese Sicht

[1] Das ist Fluch und Segen zugleich: Während diese Informationen einerseits Reflection möglich machen, bereiten sie auch den Weg zu Reverse-Engineering. Diese weitgehende Rekonstruktion von Quelltext aus Bytecode ist nicht immer erwünscht. Mit speziellen Werkzeugen, sogenannten *Code Obfuscators*, kann Bytecode unter Erhalt der Funktionalität so durcheinander gebracht werden, dass Reverse-Engineering (und damit auch Reflection) nur noch wenig brauchbare Informationen liefert.

erlaubt keine Änderungen.[2] Darüber hinaus reicht sie nur bis zur Ebene von Klassenelementen. Reflection „sieht“ also Methoden (einschließlich Konstruktoren), Variablen und geschachtelte Klassen. Methodenrümpfe, also lokale Variablen, Anweisungen, Kontrollstrukturen und dergleichen, bleiben dagegen verborgen.

Reflection macht auch den inneren Aufbau von Objekten sichtbar. Hier beschränken sich die Möglichkeiten nicht auf die bloße Besichtigung. Reflection kann auch Methoden der Objekte aufrufen oder die Werte von Objektvariablen ändern. Wohlgemerkt, es geht dabei um Methoden und Variablen, deren Existenz erst zur Laufzeit aufgespürt wurde. Als die reflektive Anwendung selbst übersetzt wurde, waren sie nicht bekannt. Nachdem zur Laufzeit kein Compiler mehr im Spiel ist, lässt sich beispielsweise der Zugriffsschutz durch `private` ohne Weiteres umgehen.

8.1 Factory-Methoden

Flexible Alternative zu Konstruktoren

Die meisten Klassen definieren Konstruktoren, um Objekte zu erzeugen. Eine Alternative dazu bieten **Factory-Methoden**. In der Bibliothek gibt einige Beispiele:

```
static Pattern  compile(String regex)      // java.util.regex.Pattern
static Path  get(String pathname, ...)     // java.nio.file.Paths
Socket  accept()                           // java.net.ServerSocket
```

Factory-Methoden können als statische oder als normale Methoden definiert sein. Dieser Unterschied ist nicht entscheidend für die Arbeitsweise.

Merkmale von Factory-Methoden

Vorteile von Factory-Methoden gegenüber Konstruktoren sind unter anderem:[3]

- Factory-Methoden können bei jedem Aufruf neu entscheiden, welche konkrete Klasse das erzeugte Objekt haben soll, wenn es nur kompatibel zum Ergebnistyp ist.
- Der Code des Anwenders muss sich auf keine konkrete Klasse festlegen. Das entkoppelt Klassen und verringert Abhängigkeiten.
- Factory-Methoden können als Ergebnis `null` zurückgeben, wenn sie kein Objekt erzeugen wollen. Konstruktoren können nur mit einer Exception scheitern.

[2] Es gibt mit den Mitteln der Laufzeitbibliothek keine Möglichkeit, zur Laufzeit bestehenden Code zu manipulieren. Ein Java-Programm kann sich nicht selbst modifizieren. Von anderen Anbietern gibt es aber durchaus solche Ansätze, wie beispielsweise die „Byte Code Engineering Library“ der Apache Foundation.

[3] Sie erfreuen sich deswegen wachsender Beliebtheit. Immer mehr Klassen der Laufzeitbibliothek nutzen Factory-Methoden.

- Nicht statische Factory-Methoden sind normale Methoden und werden dynamisch gebunden. Sie können bei Bedarf redefiniert werden, ohne dass der Aufrufer davon betroffen ist.

Reflection ist der Schlüssel zur Implementierung von Factory-Methoden. Den Einstieg in die Reflection bildet die Klasse `Class`, deren Instanzen sogenannte **Typobjekte** sind.

8.1.1 Typobjekte und die Methode `forName`

Typobjekte repräsentieren Typen

Zu jedem Typ in Java gibt es ein korrespondierendes **Typobjekt**. Das gilt für *alle* Typen, ob vordefinert oder benutzerdefiniert, ob primitiv oder Referenztyp, ob Klasse oder Interface. Typobjekte sind eindeutig, das heißt, zu jedem Typ existiert *genau ein* Typobjekt und kein zweites.

Typobjekte sind selbst Java-Objekte und haben, wie alle Objekte, einen Typ. Dieser Typ ist eine Ausprägung der generischen Klasse `Class<T>`. `Class` hat keinen öffentlichen Konstruktor. Der Anwender kann Typobjekte auch nicht direkt erzeugen. Die JVM legt sie in dem Moment an, in dem sie den Bytecode einer Klasse lädt.[4]

Wege zu Typobjekten

Zugriff auf Typobjekte erhält man auf verschiedenen Wegen:

- In `Object` ist der Getter

  ```
  final Class<?>  getClass()
  ```

  definiert.[5] Jede Klasse erbt diese Methode. Sie kann folglich mit jedem Objekt aufgerufen werden und liefert immer das Typobjekt zur Klasse des Zielobjekts.[6] Der folgende Ausdruck liefert das Typobjekt der Klasse `String`:

  ```
  "Hello".getClass()
  ```

  Im nächsten Ausdruck erzeugt die Methode `asList` eine zum Interface `List` kompatible Sicht auf das Array, deren konkreter Typ nicht bekannt ist. `getClass` liefert dennoch das betreffende Typobjekt.[7]

---

4 Der Aufruf von `java` mit dem Schalter `-verbose` macht das Laden von Klassen durch die JVM sichtbar.

5 `getClass` ist `final`, weil eine Redefinition nicht sinnvoll ist.

6 `getClass` liefert das Typobjekt des *dynamischen* Typs, also des tatsächlichen Laufzeittyps des Zielobjekts. Es repräsentiert immer eine konkrete Klasse. Dem steht der statische Typ gegenüber, den die Definition im Quelltext festlegt. Sie ist für den Compiler maßgeblich und spielt zur Laufzeit und damit für die Reflection keine Rolle.

7 Eine Untersuchung des Typobjekts zeigt, dass es sich um eine statisch geschachtelte Klasse `Arrays.ArrayList` handelt.

```
Arrays.asList(new String[] {"Hello", "Duke"}).getClass();
```

- Das Literal

  ```
  type.class
  ```

  bezeichnet das Typobjekt von *type*. Dabei kann *type* eine beliebige Typangabe sein, einschließlich ABCs, Interfaces, primitiver Typen, Arrays und des Pseudotyps `void`. Hier einige Beispiele:

  ```
  String.class
  Object.class
  Iterable.class
  int.class
  int[].class
  void.class
  ```

`forName` akzeptiert Typnamen als String

- Die statische Factory-Methode

  ```
  Class<?>  forName(String name)
  ```

  sucht das Typobjekt des Typs mit dem Namen *name*.[8] Wenn sie keines findet, wirft sie eine `ClassNotFoundException`. Der erste der beiden folgenden Ausdrücke liefert das Typobjekt von `String`, der zweite wirft eine Exception:

  ```
  Class.forName("java.lang.String")
  Class.forName("Hello, world!")
  ```

Das folgende Programm demonstriert die Zugriffswege. An alle drei Variablen wird das*selbe* Typobjekt zugewiesen, nämlich das Typobjekt der Klasse `String`. Das Programm gibt deshalb `true` aus.

```
public class Typeobjects {
    public static void main(String... args) throws ClassNotFoundException {
        Class<?> fromObject = "Hello".getClass();
        Class<?> literal = String.class;
        Class<?> lookedUp = Class.forName("java.lang.String");
        System.out.println(fromObject == literal && literal == lookedUp);
    }
}
```

Listing 8.1: Zugriff auf Typobjekte auf verschiedenen Wegen.

Meta-Klasse `Class`

`Class` ist eine generische Klasse. Das Typargument ist im Einzelfall genau der Typ, den das Typobjekt repräsentiert. Beispielsweise hat das Typobjekt der Klasse `String`

[8] Das Argument muss einen qualifizierten Namen nennen, also eine Typangabe einschließlich des kompletten Package-Pfads. `import`-Klauseln sind wirkungslos. Selbst der sonst automatische Import von `java.lang` wird nicht angewendet.

den generischen Typ `Class<String>`. Weil Typobjekte eindeutig sind, gibt es kein zweites Objekt mit dem generischen Typ `Class<String>`.

`Class` wird als **Meta-Klasse** bezeichnet, weil ihre Objekte wiederum Typen repräsentieren. Das Typobjekt von `Class` selbst hat den Typ `Class<Class<?>>`.

Die Variablen in `Typeobjects` (Listing 8.1) sind mit Wildcards definiert und akzeptieren daher `Class`-Werte mit beliebigen Typargumenten. Man kann die Variablen auch mit dem konkreten generischen Typ `Class<String>` definieren, muss dann aber Typecasts einfügen, weil `getClass` und `forName` nur den Typ `Class<?>` liefern. Beim Literal `String.class` ist das nicht nötig, weil der Compiler den konkreten generischen Typ erkennt. Am Ergebnis ändert das nichts.

```
Class<String> fromObject = (Class<String>)"Hello".getClass();
Class<String> literal = String.class;
Class<String> lookedUp = (Class<String>)Class.forName("java.lang.String");
```

Listing 8.2: Typecasts für `getClass` und `forName`.

8.1.2 Erzeugen von Objekten mit `newInstance`

Konstruktoraufruf über Typobjekt

Die statische Factory-Methode `forName` der Klasse `Class` liefert ein Typobjekt. Ausgehend von einem solchen Typobjekt erzeugt die Methode

```
T  newInstance()
```

ein Objekt der betreffenden Klasse. Ein Aufruf von `newInstance` entspricht dem Aufruf des Default-Konstruktors der betreffenden Klasse. Das neue Objekt hat den Typ `T`, das Typargument des Typobjekts. Das folgende Codefragment produziert beispielsweise einen neuen, leeren String:

```
Class<String> cs = String.class;
String s = cs.newInstance();
```

Fehlerquellen beim Instanziieren unbekannter Klassen

Es gibt eine ganze Reihe von Gründen, aus denen `newInstance` scheitern kann. Diese äußern sich in verschiedenen Exceptions:[9]

[9] Zu den hier gezeigten Exceptions kommt noch die `SecurityException`, die aber in ein ganz anderes Minenfeld führt.

`InstantiationException`
: Der Typ kennt keinen Default-Konstruktor. Er ist beispielsweise primitiv, ein Interface, eine abstrakte Basisklasse[10] oder ein Array.

`IllegalAccessException`
: Entweder die Klasse oder der Default-Konstruktor sind nicht zugänglich.

`ExceptionInInitializerError`
: Der Default-Konstruktor wurde aufgerufen, warf aber eine Exception.

Zur Vereinfachung des Umgangs mit `newInstance` und anderen Methoden, die mit Reflection arbeiten, gibt es die gemeinsame Basisklasse `ReflectiveOperationException`, unter der alle Fehlerquellen im Zusammenhang mit Reflection gesammelt sind. Dieser Exceptiontyp reicht aus, wenn keine differenzierte Reaktion erforderlich ist.

8.1.3 Beispielanwendung: Pizzas

Als Beispiel sollen Pizzas dienen, die in Anhang C zur Illustration des Decorator-Patterns eingeführt werden. Hier ein Überblick über die dabei definierten Typen:

`interface Pizza`
: Gemeinsames Interface aller Pizzas.

`abstract class Base implements Pizza`
: ABC für alle Arten von Pizzaböden.

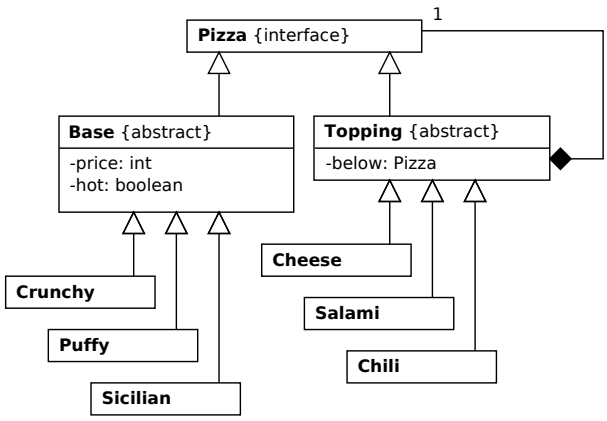
`class Crunchy extends Base`
: Konkreter Pizzaboden, ebenso `Puffy` und `Sicilian`.

`abstract class Topping implements Pizza`
: ABC für alle Arten von Pizzaauflagen.

`class Cheese extends Topping`
: Konkrete Auflage Käse, ebenso `Salami` und `Chili`.

Das folgende Diagramm zeigt die Beziehungen zwischen den Typen (Kopie von Seite 626):

[10] Eine ABC mag einen Default-Konstruktor haben, allerdings steht er nur Konstruktoren abgeleiteter Klassen zur Verfügung.



Die Anwendung `PizzaMain` baut eine Pizza gemäß Kommandozeilenargumenten zusammen und berechnet die Eigenschaften dieser Pizza. `PizzaMain` funktioniert zwar, ist aber nicht gerade ein Ausbund an Flexibilität. Sie enthält insbesondere eine fest codierte Abbildung von Eingabestrings auf die bekannten `Base`- und `Topping`-Klassen.

Konkretes Beispiel einer Factory-Methode

Factory-Methoden erlauben eine weit bessere Fassung von `PizzaMain`. Statt fest codierter Strings (`"Crunchy"` für einen `Crunchy`-Boden und so weiter) akzeptiert die Methode `makeBase` den Namen einer beliebigen `Base`-Klasse und erzeugt daraus ein passendes `Base`-Objekt. `makeBase` wird in die getrennte Utility-Klasse `PizzaFactory` ausgelagert:

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

public class PizzaFactory {
    public static Base makeBase(String typeName) throws ReflectiveOperationException {
        Class<Base> type = (Class<Base>)Class.forName(typeName);
        return type.newInstance();
    }
}
```

Listing 8.3: Utility-Klasse mit Factory-Methoden für Pizzabestandteile.

Die Anwendung ruft nun diese Factory-Methode auf, um Pizzaböden zu produzieren:

```
for(String arg: args)
    if(pizza == null)
        pizza = PizzaFactory.makeBase(arg);
    else
        switch(arg) {
```

```
            case "Cheese":
                pizza = new Cheese(pizza);
                break;
            case "Salami":
                pizza = new Salami(pizza);
                break;
            case "Chili":
                pizza = new Chili(pizza);
                break;
        }
```

Listing 8.4: Factory-Methode zum Produzieren von Pizzaböden.

Äußerlich arbeitet das Programm unverändert:

```
$ java PizzaBetterMain Crunchy Cheese Cheese Salami Chili
Your pizza:
price: 650
vegetarian: false
hot: true
```

Entkopplung von konkreten Klassennamen

Diese Fassung ist flexibler als `PizzaMain`, weil sie keine konkreten Typen von Pizzaböden mehr kennt und unabhängig von diesen ist. Änderungen in der Auswahl der Pizzaböden berühren die Anwendung nicht mehr. Sie funktioniert ohne Neu-Übersetzung weiter und kommt beispielsweise mit neuen Pizzaböden sofort zurecht.

8.1.4 Konstruktor-Objekte

Typen für Bausteine von Klassen

`Class`-Objekte repräsentieren Klassen, die Bausteine von Java-Programmen. Auch für die Komponenten von Klassen gibt es Typen mit selbsterklärenden Namen:

- `Method`
- `Field` (Objekt- oder Klassenvariable)
- `Constructor`
- `Modifier`

Diese Typen sind, anders als `Class`, im Package `java.lang.reflect` definiert und nicht in `java.lang`.[11] Der nächste Abschnitt geht genauer auf die einzelnen Klassen

[11] Dafür gibt es verschiedene Gründe. Unter anderem soll das Package `java.lang` nicht mit Typen aufgebläht werden, die eher selten gebraucht werden.

ein. Hier wird der Typ `Constructor` herausgegriffen, der für Factory-Methoden interessant ist.

Aufruf von Custom-Konstruktoren

Die Factory-Methode `makeBase` in `PizzaFactory` (Listing 8.3) erzeugt neue `Base`-Objekte mithilfe der Methode `newInstance`, die einen Default-Konstruktor aufruft. Für `Base`-Klassen passt das gut, weil sie alle einen Default-Konstruktor definieren.

`Topping`-Objekte haben dagegen keine Default-Konstruktoren. Als „Dekoratoren" im Decorator-Pattern *müssen* sie mit einer bereits existierenden Pizza initialisiert werden und sind isoliert weder sinnvoll noch lebensfähig. Gesucht ist ein Weg, um genau den Custom-Konstruktor eines `Topping`-Typobjekts aufzurufen, der einen `Pizza`-Parameter akzeptiert.

Klasse `Constructor` repräsentiert einen Konstruktor

Zu diesem Zweck definiert `Class` die Methode

```
Constructor<T>  getConstructor(Class<?>... types)
```

Diese Methode lokalisiert den einen Konstruktor, dessen Parameterliste aus den Typen `types` besteht. `types` ist ein Vararg-Parameter, weil Konstruktoren beliebig lange Parameterlisten haben können. `getConstructor` liefert ein Ergebnis vom Typ `Constructor<T>` zurück, wobei das Typargument `T` mit dem Typargument des Zielobjekts übereinstimmt. Das von `getConstructor` zurückgegebene Objekt „ist ein" Konstruktor. Das folgende Codefragment lokalisiert zum Beispiel unter den verschiedenen String-Konstruktoren den einen, der ein Array von Zeichen akzeptiert:

```
Class<String> type = String.class;
Constructor<String> ctor = type.getConstructor(char[].class);
```

Die Variable `ctor` „enthält" jetzt den String-Konstruktor mit der Signatur `String(char[])` als Wert.

Aufruf eines Konstruktor-Objekts

Der einzige Bestimmungszweck eines Konstruktors ist es, aufgerufen zu werden. Dazu dient die `Constructor`-Methode[12]

```
T  newInstance(Object... args)
```

Die Anzahl und die Typen der Argumente für `newInstance` müssen zu den Argumenten des vorangegangenen `getConstructor`-Aufrufs passen. Der String-Konstruktor `ctor` im oben gezeigten Codefragment kann also mit

```
String string = ctor.newInstance(new char[] {'H', 'e', 'l', 'l', 'o'});
```

[12] Es gibt in der Klasse `Class` und in der Klasse `Constructor` jeweils eine Methode namens `newInstance`. Erstere ist parameterlos, Letztere hat einen Parameter. Das sind zwei *verschiedene* Methoden mit dem gleichen Namen und mit ähnlichen Aufgaben.

aufgerufen werden[13] und produziert dann den String `"Hello"`.

8.1.5 Beispielanwendung: Pizzas, weitere Verbesserung

Factory-Methode mit Konstruktor-parametern

Mit diesem Rüstzeug kann in der Klasse `PizzaFactory` (Listing 8.3) eine weitere statische Factory-Methode definiert werden, die eine Pizzaauflage erzeugt und eine gegebene Pizza damit dekoriert:

```
public static Topping makeTopping(String typeName, Pizza pizza) throws ReflectiveOperatio
    Class<Topping> type = (Class<Topping>)Class.forName(typeName);
    Constructor<Topping> ctor = type.getConstructor(Pizza.class);
    return ctor.newInstance(pizza);
}
```

Listing 8.5: Statische Factory-Methode mit Aufruf eines Custom-Konstruktors.

Zusammen mit der vorhergehenden Factory-Methode der `PizzaFactory` (Listing 8.3) kann `PizzaBetterMain` zu einer neuen Anwendung `PizzaDeluxeMain` vereinfacht werden, die komplett von konkreten Klassennamen entkoppelt ist:

```
for(String arg: args)
    if(pizza == null)
        pizza = PizzaFactory.makeBase(arg);
    else
        pizza = PizzaFactory.makeTopping(arg, pizza);
```

Listing 8.6: Produzieren von Pizzas mit Factory-Methoden.

`PizzaDeluxeMain` arbeitet aus Sicht des Anwenders wie gehabt:

```
$ java PizzaDeluxeMain Crunchy Cheese Cheese Salami Chili
Your pizza:
price: 650
vegetarian: false
hot: true
```

Robust gegenüber Änderungen

Diese Fassung ist robust gegenüber Änderungen. Beispielsweise werden nachträglich „italienische Kräuter“ als neue Auflage und „Mama Leone“ als neuer Pizzaboden ins Angebot aufgenommen:

[13] Ein Typecast ist nicht nötig, weil `ctor` den Typ `Constructor<String>` hat und folglich einen `String` liefert.

```
public class ItalianHerbs extends Topping {
    public ItalianHerbs(Pizza below) {
        super(below);
    }
}
```

Listing 8.7: Neue Pizzaauflage.

```
public class MamaLeone extends Base {
    public MamaLeone() {
        super(500, false);
    }
}
```

Listing 8.8: Voluminöser Pizzaboden.

Die Anwendung kommt ohne weitere Anpassungen damit zurecht:

```
$ java PizzaDeluxeMain MamaLeone Cheese Cheese Salami ItalianHerbs
Your pizza:
price: 850
vegetarian: false
hot: false
```

8.2 Analyse der Codestruktur

Getter liefern Bausteine einer Klassendefinition

Die weiter vorne (Seite 534) angesprochenen Klassen repräsentieren verschiedene Bausteine eines Java-Programms. `Class` und `Package` sind als einzige dieser Klassen im Package `java.lang` definiert. Alle anderen sind Teil des Packages `java.lang.reflect`. Die Möglichkeiten zum Zugriff auf Typobjekte wurden bereits gezeigt (Seite 529).

Grundsätzliche Auskunft über die Art des Typs, für den ein `Class`-Objekt steht, geben die folgenden Prädikate. Die Namen sprechen für sich:

```
boolean  isPrimitive()
boolean  isArray()
boolean  isEnum()
boolean  isInterface()
boolean  isAnonymousClass()
boolean  isLocalClass()
boolean  isMemberClass()
```

Start mit einem Typobjekt

Ausgehend vom `Class`-Objekt lassen sich damit sukzessive Einzelheiten einer Typdefinition ergründen:

- Name und andere Meta-Informationen der Klasse:

  `String getName()`
  : Qualifizierter Name dieses Typs (entsprechend `getSimpleName`: Name ohne Package-Pfad).

  `Package getPackage()`
  : Package, in dem dieser Typ definiert ist.

  `int getModifiers()`
  : Modifier dieser Typdefinition.

- Kompatibilität zu anderen Typen:

  `Class<?>[] getInterfaces()`
  : Alle Interfaces, die dieser Typ implementiert.

  `Class<? super T> getSuperclass()`
  : Basisklasse oder `null`, wenn es keine gibt.

  `boolean isAssignableFrom(Class<?> type)`
  : Ist der Typ `type` zuweisungskompatibel zu diesem Typ?

  `boolean isInstance(Object x)`
  : Ist `x` von diesem Typ? Dieser Test entspricht dem Vergleich mit `instanceof`.

- Konstruktoren:

  `Constructor<T> getConstructor(Class<?>... types)`
  : Konstruktor mit den Parametertypen `types` oder `NoSuchMethodException` (entsprechend `getDeclaredConstructor`: Konstruktor laut Quelltext).

  `Constructor<T>[] getConstructors()`
  : Alle öffentlichen Konstruktoren (entsprechend `getDeclaredConstructors`: alle Konstruktoren laut Quelltext).

- Variablen:

  `Field getField(String name)`
  : Öffentliche Objekt- oder Klassenvariable mit dem Namen `name` (entsprechend `getDeclaredField`: Objekt- oder Klassenvariable laut Quelltext).

  `Field[] getFields()`
  : Alle öffentlichen Objekt- und Klassenvariablen (entsprechend `getDeclaredFields`: alle Objekt- und Klassenvariablen laut Quelltext).

- Methoden:

  `Method getMethod(String name, Class<?>... type)`
  Öffentliche Methode namens `name` mit Parametertypen `types` (entsprechend `getDeclaredMethod`: Methode laut Quelltext).

  `Method[] getMethods()`
  Alle öffentlichen Methoden (entsprechend `getDeclaredMethods`: Methoden laut Quelltext).

Eigenschaften von Arrays und Enums

Speziell bei Array- und Aufzählungstypen sind darüber hinaus interessant:

`Class<?> getComponentType()`
Der Elementtyp bei einem Array oder `null` ansonsten.

`T[] getEnumConstants()`
Elemente bei einem Aufzählungstyp oder `null` ansonsten.

8.2.1 Modifier

Auskunft über Modifier einer Definition

Die Methode `getModifiers` liefert die Modifier einer Typdefinition als `int`-Wert, dessen Bits einzelnen Modifiern entsprechen. Die Klasse `Modifier` definiert entsprechende Masken, in denen jeweils ein Bit gesetzt ist, als öffentliche Konstanten, wie zum Beispiel `Modifier.PUBLIC`. Bequemer geben einfache Bitfilter-Methoden Auskunft:

```
static boolean  isAbstract(int modifiers)
static boolean  isFinal(int modifiers)
static boolean  isInterface(int modifiers)
static boolean  isNative(int modifiers)
static boolean  isPrivate(int modifiers)
static boolean  isProtected(int modifiers)
static boolean  isPublic(int modifiers)
static boolean  isStatic(int modifiers)
static boolean  isStrict(int modifiers)
static boolean  isSynchronized(int modifiers)
static boolean  isTransient(int modifiers)
static boolean  isVolatile(int modifiers)
```

Beispielsweise gibt der folgende Aufruf Auskunft darüber, dass die Klasse `String` nicht abgeleitet werden kann:

```
Modifier.isFinal(String.class.getModifiers()) → true
```

Modifier-Mengen bestimmter Arten von Definitionen

Nicht alle Modifier sind für jede Art von Definition überhaupt sinnvoll. Die folgenden Methoden geben jeweils einen konstanten `int`-Wert zurück, in dem genau die Bits der Modifier gesetzt sind, die für die betreffende Definition zulässig sind:

```
static int  classModifiers()
static int  interfaceModifiers()
static int  constructorModifiers()
static int  fieldModifiers()
static int  methodModifiers()
```

Eine lesbare Darstellung einer Sammlung von Modifiern produziert die statische Methode

```
static String Modifier.toString(int modifiers)
```

Man erhält zum Beispiel:

```
Modifier.toString(Modifier.classModifiers()) → "public protected private abstract static
Modifier.toString(int[].class.getModifiers()) → "public abstract final"
```

8.2.2 Konstruktoren

Zugriff auf und Auswahl von Konstruktoren

Die Methoden

```
Constructor<T>  getConstructor(Class<?>... types)
Constructor<T>  getDeclaredConstructor(Class<?>... types)
Constructor<T>[]  getConstructors()
Constructor<T>[]  getDeclaredConstructors()
```

liefern einen oder mehrere Konstruktoren eines Typs, sofern es überhaupt Konstruktoren gibt. Die `Declared`-Varianten beziehen sich auf die Typdefinition ohne Rücksicht auf Zugriffsschutz-Modifier, die anderen Methoden erfassen nur öffentliche (`public`-)Konstruktoren.

Nähere Angaben über einen Konstruktor

Eine Reihe von Gettern liefert weitere Informationen zu einem Konstruktor:

```
Class<?>[]  getExceptionTypes()
int  getModifiers()
Class<?>[]  getParameterTypes()
boolean  isVarArgs()
```

Das folgende Programm druckt mithilfe dieser Methode Informationen über die Konstruktoren einer Klasse aus, deren qualifizierter Name auf der Kommandozeile angegeben ist:

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

public class PrintCtorDetails {
    public static void main(String... args) throws ReflectiveOperationException {
        Class<?> type = Class.forName(args[0]);
        for(Constructor<?> ctor: type.getDeclaredConstructors()) {
            out.println("Ctor:");
            out.printf("\tModifiers: %s%n", Modifier.toString(ctor.getModifiers()));

            out.print("\tParameters: ");
            for(Class<?> paramType: ctor.getParameterTypes())
                out.printf("%s ", paramType.getSimpleName());
            out.printf("%s%n", ctor.isVarArgs()? "...": "");

            out.print("\tExceptions: ");
            for(Class<?> exceptionType: ctor.getExceptionTypes())
                out.printf("%s ", exceptionType.getSimpleName());
            out.println();
        }
    }
}
```

Listing 8.9: Ausgabe von Informationen über die Konstruktoren eines Typs.

Die Klasse selbst verfügt nur über einen Konstruktor, nämlich den vom Compiler automatisch ergänzten Default-Konstruktor:

Beispiel: automatisch definierter Default-Konstruktor

```
$ java PrintCtorDetails PrintCtorDetails
Ctor:
        Modifiers: public
        Parameters:
        Exceptions:
```

Reichhaltiger ist der Ergebnis beim Aufruf mit `java.lang.String`, das hier der Länge wegen nicht abgedruckt ist.

8.2.3 Variablen

Variablen einer Typdefinition erhält man als `Field`-Objekte mit den `Class`-Methoden:[14]

Ermitteln der Klassen- und Objektvariablen

```
Field  getField(String name)
Field  getDeclaredField(String name)
Field[]  getFields()
Field[]  getDeclaredFields()
```

[14] Die `Declared`-Varianten ignorieren Zugriffsschutz-Modifier.

Einige Getter liefern genauere Informationen zu einer Variablen:

```
int  getModifiers()
String  getName()
Class<?>  getType()
boolean  isEnumConstant()
```

8.2.4 Methoden

Liste der Methoden

Auskunft über die Methoden eines Typs (ohne die Konstruktoren) liefern die folgenden Methoden in Form von Objekten des Typs `Method`:

```
Method  getMethod(String name, Class<?>... type)
Method  getDeclaredMethod(String name, Class<?>... type)
Method[]  getMethods()
Method[]  getDeclaredMethods()
```

Die Bestandteile der Signatur einer Methode erfährt man mit:

```
int  getModifiers()
String  getName()
Class<?>[]  getParameterTypes()
boolean  isVarArgs()
Class<?>  getReturnType()
Class<?>  getExceptionTypes()
```

Die drei Klassen `Constructor`, `Field` und `Method` erben von der gemeinsamen Basisklasse `AccessibleObject`, die beim reflektiven Zugriff auf Objekte eine Rolle spielt (Seite 546).

8.2.5 Anwendung: UML-artige Skizze einer Klassendefinition

Informationen in statischen Klassendiagrammen

Statische Klassendiagramme der UML[15] zeigen Beziehungen zwischen Typen und ausgewählte Informationen darüber in einer halbgrafischen Darstellung. Typen sind als Boxen mit bis zu drei „Schüben“ für Namen, Variablen und Methoden dargestellt. In den drei Schüben steht Text mit dem folgenden Aufbau:

1. Name des Typs mit optionalem Stereotyp und optionaler Markierung als abstrakt:

   `[<<interface>>]` *`name`* `[{abstract}]`

[15] Siehe etwa `http://www.omg.org/spec/UML/ISO/19501/PDF`, Seite 201 ff.

2. Liste der Variablen in der Form:

   name: *type*

3. Liste der Methoden in der Form:

   name(*paramname*: *paramtype*, ...): *returntype* [{abstract}]

Zusätzlich sind Klassenvariablen und statische Methoden unterstrichen. Ein vorangestelltes Zeichen drückt die Sichtbarkeit aus (+ = öffentlich, - = privat, # = für abgeleitete Klassen, keine Angabe = Voreinstellung). Viele weitere Details der UML bleiben hier zur Vereinfachung unberücksichtigt.

Textdarstellung einer einzelnen Typdefinition

Das folgende Programm gibt eine Textdarstellung der Klasseninformationen in Anlehnung an die oben beschriebene Form auf dem Bildschirm aus. Abweichend gilt für die Textausgabe:

- Ein Unterstrich am Zeilenanfang und -ende dient als Ersatz für Unterstreichung.
- Abstrakte Elemente werden in Schrägstriche / eingefasst und nicht mit `{abstract}` markiert.
- Aufzählungstypen werden mit dem Stereotyp `<<enum>>` gekennzeichnet.

Statische Hilfsmethoden erledigen wiederkehrende Aufgaben.

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

public class UMLClassBox {
    public static void main(String... args) throws ReflectiveOperationException {
        for(String name: args) {
            final Class<?> type = Class.forName(name);
            out.println(umlStereoType(type)
                                + umlAttributes(type.getSimpleName(), type.getModifiers())
            out.println("--");
            for(Field field: type.getDeclaredFields())
                out.println(umlModifierString(field.getModifiers())
                                + field.getName()
                                + ": "
                                + field.getType().getSimpleName());
            out.println("--");
            for(Constructor<?> ctor: type.getDeclaredConstructors())
                out.println(umlModifierString(ctor.getModifiers())
                                + type.getSimpleName()
                                + umlParameters(ctor.getParameterTypes()));
            for(Method method: type.getDeclaredMethods())
                out.println(umlAttributes(umlModifierString(method.getModifiers())
                                                + method.getName()
                                                + umlParameters(method.getParameterTypes
                                                + ": "
                                                + method.getReturnType().getSimpleName()
```

```
                                                    method.getModifiers()));
            out.println();
        }
    }

    private static String umlModifierString(int modifiersMask) {
        String result = "";
        if(Modifier.isPrivate(modifiersMask))
            return result + "-";
        if(Modifier.isPublic(modifiersMask))
            return result + "+";
        if(Modifier.isProtected(modifiersMask))
            return result + "#";
        return result;
    }

    private static String umlStereoType(Class<?> type) {
        if(type.isInterface())
            return "<<interface>>\n";
        if(type.isEnum())
            return "<<enum>>\n";
        return "";
    }

    private static String umlAttributes(String name, int modifiersMask) {
        String result = name;
        if(Modifier.isAbstract(modifiersMask))
            result = "/" + result + "/";
        if(Modifier.isStatic(modifiersMask))
            result = "_" + result + "_";
        return result;
    }

    private static String umlParameters(Class<?>[] paramTypes) {
        String result = "";
        for(Class<?> type: paramTypes)
            result += ", " + type.getSimpleName();
        if(!result.isEmpty())
            result = result.substring(2);
        return "(" + result + ")";
    }
}
```

Listing 8.10: Ausgabe einer UML-artigen Textdarstellung von Klasseninformationen.

Der folgende Aufruf gibt einen Überblick über `UMLClassBox` (Listing 8.10) selbst:

```
$ java UMLClassBox UMLClassBox
UMLClassBox
--
--
+UMLClassBox()
_+main(String[]): void_
_-umlModifierString(int): String_
_-umlStereoType(Class): String_
```

```
_-umlAttributes(String, int): String_
_-umlParameters(Class[]): String_
```

Interessante Einzelheiten liefert beispielsweise der Aufruf mit `java.lang.String`. Der Kürze wegen ist das Ergebnis hier nicht abgedruckt.

8.3 Analyse und Modifikation von Objekten

Zugriff auf Objekte via Reflection

Die Mittel des vorhergehenden Abschnitts erlauben die Untersuchung von Typen und liefern zum Beispiel Einzelheiten über die Innenstruktur einer Klassendefinition. Sie beziehen ihre Informationen aus dem Bytecode, zeigen also eine Sicht auf den Code eines Programms.

Die Möglichkeiten der Reflection gehen aber weiter. In diesem Abschnitt geht es um die reflektive Untersuchung und Manipulation von Objekten, also den Daten des laufenden Programms.

8.3.1 Zugriff auf Variablen

Auslesen der Werte von Variablen

Die Klasse `Field` definiert, abgesehen von den auf Seite 541 gezeigten Gettern, einen Satz von Methoden, mit denen der Wert der Variablen eines gegebenen Objekts `x` ausgelesen werden kann. Bei Klassenvariablen wird `x` ignoriert.

```
Object  get(Object x)
```

`get` packt primitive Werte in Wrapper-Objekte. Die folgenden Methoden umgehen das und liefern primitive Werte direkt zurück:

```
boolean  getBoolean(Object x)
byte  getByte(Object x)
char  getChar(Object x)
double  getDouble(Object x)
float  getFloat(Object x)
int  getInt(Object x)
long  getLong(Object x)
short  getShort(Object x)
```

Diese Methoden unterliegen den Zugriffsregeln von Java. Zum Beispiel erlauben sie keinen Zugriff auf private Variablen von außerhalb der Klasse.

Freigabe des Zugriffs

`private` wirkungslos bei Reflection

Der Zugriffsschutz von Variablen kann explizit außer Kraft gesetzt werden. Die beiden folgenden Methoden in `AccessibleObject`, der Basisklasse von `Field`, arbeiten mit dem entsprechenden Schalter:

```
boolean  isAccessible()


void  setAccessible(boolean flag)
```

Diese Maßnahme ist auf den Zugriff via Reflection beschränkt und hat keine Auswirkungen auf anderen Code. Sie greift auch bei Konstruktoren und Methoden, weil `Constructor` und `Method` ebenfalls von `AccessibleObject` abgeleitet sind.

Die folgende „verschlossene“ Klasse mit privaten Objektvariablen dient als Beispiel:

```
class ClosedThing {
    private String name;

    private int size;

    public ClosedThing(String name, int size) {
        this.name = name;
        this.size = size;
    }
}
```

Listing 8.11: Klasse mit privaten Objektvariablen.

Das folgende Programm erzeugt zwei `ClosedThing`-Objekte und listet ihre Objektvariablen samt Werten auf:

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

public class DumpVariables {
    public static void main(String... args) throws ReflectiveOperationException {
        dumpVariables(new ClosedThing("small", 1));
        dumpVariables(new ClosedThing("large", 5));
        dumpVariables("Hello");
    }

    static void dumpVariables(Object x) throws ReflectiveOperationException {
        System.out.println(x);
        for(Field field: x.getClass().getDeclaredFields()) {
```

```
            field.setAccessible(true);
            System.out.printf("\t%s %s = %s%n",
                              field.getType().getSimpleName(),
                              field.getName(),
                              field.get(x));
        }
    }
}
```

Listing 8.12: Private Objektvariablen von Objekten auslesen und ausgeben.

Das Programm gibt aus:

```
$ java DumpVariables
ClosedThing@1fa157c
        String name = small
        int size = 1
ClosedThing@cdc97b
        String name = large
        int size = 5
Hello
        char[] value = [C@1a0d111
        int offset = 0
        int count = 5
        int hash = 0
        long serialVersionUID = -6849794470754667710
        ObjectStreamField[] serialPersistentFields = [Ljava.io.ObjectStreamField;@fde8da
        Comparator CASE_INSENSITIVE_ORDER = java.lang.String$CaseInsensitiveComparator@1c
```

Ohne den `setAccessible`-Aufruf stürzt `DumpVariables` (Listing 8.12) mit einer `IllegalAccessException` ab.

8.3.2 Umgang mit Arrays

Reflektiver Zugriff auf Array-Elemente

Arrays sind zwar Referenztypen, aber keine Klassen. Die `Field`-Methoden des vorhergehenden Abschnitts eignen sich daher nicht zum Zugriff auf den Inhalt. Die Utility-Klasse `Array` im Package `java.lang.reflect` [16] definiert einen Satz statischer Methoden, mit denen Array-Elemente reflektiv erreicht werden können:

```
static Object  get(Object array, int index)
static boolean   getBoolean(Object array, int index)
static byte  getByte(Object array, int index)
static char  getChar(Object array, int index)
static double  getDouble(Object array, int index)
static float  getFloat(Object array, int index)
```

[16] Diese Klasse ist nicht zu verwechseln mit `java.util.Arrays`.

```
static int  getInt(Object array, int index)
static long  getLong(Object array, int index)
static short  getShort(Object array, int index)
```

Die erste Methode (`get`) packt primitive Ergebnisse in Wrapper-Objekte, wie die entsprechende `Field`-Methode. Die acht weiteren Getter vermeiden diesen Schritt.

Der Aufruf von

```
static int  getLength(Object array)
```

entspricht dem Ausdruck *array*`.length` in statischem Code.

Beispiel: Lesbare Darstellung eines Arrays

Die statische Hilfsmethode `dumpArray` im folgenden Programm druckt eine lesbare Darstellung des übergebenen Arguments aus, wenn es ein Array ist:

```
import static java.lang.Math.*;
import static java.lang.System.*;
import java.lang.reflect.*;

public class DumpArrays {
    public static void main(String... args) throws ReflectiveOperationException {
        dumpArray(args);
        dumpArray(new double[] {1, sqrt(2), PI});
        dumpArray("Hello");
    }

    static void dumpArray(Object x) throws ReflectiveOperationException {
        Class<?> type = x.getClass();
        if(type.isArray()) {
            out.print(type.getComponentType().getSimpleName()
                              + "["
                              + Array.getLength(x)
                              + "] = {");
            for(int i = 0; i < Array.getLength(x); i++)
                out.print((i > 0? ", ": "") + Array.get(x, i));
            out.println("}");
        }
        else
            out.println(x + ": not an array");
    }
}
```

Listing 8.13: Beliebiges Array mit Reflection analysieren und in lesbarer Form ausgeben.

Ein Aufruf zeigt die Elemente von Arrays:

```
$ java DumpArrays abra ka dabra
String[3] = {abra, ka, dabra}
double[3] = {1.0, 1.4142135623730951, 3.141592653589793}
Hello: not an array
```

8.3.3 Verändern von Objekten

Manipulation von Objekten via Reflection

Abgesehen von der Analyse können Variablen im laufenden Programm auch verändert werden. Die `Field`-Methode

```
void  set(Object x, Object value)
```

ersetzt den Wert der Variablen im Objekt `x` durch den Wert `value`. Automatisches Unboxing erlaubt bei primitiven Variablen Wrapper-Objekte als Werte. Wie bei `get` gibt es für jeden primitiven Typ *type* eine weitere Methode

```
void  setType(Object x, type value)
```

die ohne Wrapper-Objekte arbeitet.

Entsprechend ersetzen die `Array`-Methoden

```
void  set(Object array, int index, Object value)
void  setType(Object array, int index, type value)
```

das Element am Index `index` im Array `array` durch den neuen Wert `value`. Zur allgemeinen Methode `set` kommen wieder acht Methoden `set`*Type* für die primitiven Typen.

Der Zugriff auf private Variablen muss mit `setAccessible` freigeschaltet werden. Dieser Aufruf erlaubt auch Zuweisungen an Objektvariablen, die mit `final` schreibgeschützt sind. Diese scheinbar fragwürdige Freiheit nutzt zum Beispiel die Deserialisierung (Kapitel 2), um Objekte aus einer externen Darstellung zu rekonstruieren.

`final` stoppt Reflection bei Klassenvariablen

`setAccessible` wirkt *nicht* auf `final`-Klassenvariablen. Solche Variablen spielen die Rolle von Konstanten. Sie werden vom Compiler wie Code behandelt und bleiben von den regulären Objekten auf dem Heap getrennt. Code ist aus guten Gründen unveränderlich und damit auch der Manipulation durch Reflection entzogen.

8.3.4 Anwendung: Reflektive `toString`-Methode

Mit den Methoden zur reflektiven Analyse eines Objekts soll eine Methode definiert werden, die in einfachen Fällen `toString` ersetzen kann.

Viele `toString`-Implementierungen produzieren einen String der Form

```
Type(name=value, ...)
```

der mit dem Typ des Objekts beginnt und danach die Namen von Variablen mit ihren Werten auflistet. Eine statische Methode kann einen solchen String auch durch Analyse eines Objekts mit Reflection konstruieren.

Die nachfolgende Definition von `toString` delegiert die Aufgabe an eine private Hilfsmethode, die als Argumente ein Objekt und das entsprechende Typobjekt erwartet.

```
public static String toString(Object x) throws ReflectiveOperationException {
    return toString(x, x.getClass());
}
```

Listing 8.14: Öffentliche Startermethode für private, rekursive Hilfsmethode.

Die private Methode erledigt die eigentliche Arbeit:

```
private static String toString(Object x, Class<?> type) throws ReflectiveOperationExcepti
    // Einfache Fälle aussortieren
    if(x == null)                               // Objekt
        return "null";
    if(type == null || type == Object.class)    // kein Typ oder Object
        return "";
    if(type.isArray())                          // Arrays
        return Arrays.toString((Object[])x);
    if(type.isEnum() || type == String.class)   // Enums und Strings
        return x.toString();
    // String-Darstellung mit Basisklassenobjekt initialisieren
    String string = toString(x, type.getSuperclass());
    // Objekt- und Klassenvariablen durchlaufen ...
    for(Field field: type.getDeclaredFields()) {
        field.setAccessible(true);
        // Konstanten tragen nichts bei, weglassen
        int modifiers = field.getModifiers();
        if(Modifier.isStatic(modifiers) && Modifier.isFinal(modifiers))
            continue;
        Object value = field.get(x);            // Wert der Variablen
        String valueString = "";                // Wert in String-Darstellung
        if(value == null)
            valueString = "null";
        else if(field.getType().isPrimitive() || field.getType() == String.class)
            valueString = value.toString();     // Primitive Werte und Strings einfügen
        else
            valueString = value.getClass().getSimpleName(); // Bei Referenzvariablen reic
        string += ", " + field.getName() + "=" + valueString;
    }
    // Klassennamen und Variablenwerte zusammenfügen
    if(string.startsWith(", "))
        string = string.substring(2);
    return type.getSimpleName() + "(" + string + ")";
```

```
}
```

Listing 8.15: Reflektive `toString`-Methode für beliebige Objekte.

Die folgende `main`-Methode druckt einige willkürlich zusammengestellte Objekte aus (`toString` ist statisch aus `ReflectiveToString` importiert):

```
public static void main(String[] args) throws ReflectiveOperationException {
    out.println(toString(args[0]));
    out.println(toString(args));
    out.println(toString(new Puffy()));
    out.println(toString(new Cheese(new Puffy())));
    out.println(toString(Pattern.compile("a*b")));
    out.println(toString(new ReflectiveToString()));
}
```

Listing 8.16: Test der reflektiven `toString`-Methode mit verschiedenen Objekten.

Die Ergebnisse sind durchaus akzeptabel. Das `ReflectiveToString`-Objekt, das als Letztes ausgegeben wird, hat keine brauchbaren Merkmale. Man erhält daher eine etwas unspezifische Textdarstellung, die bloß aus der Typangabe besteht:

```
$ java ReflectiveToString abra ka dabra
abra
[abra, ka, dabra]
Puffy(Base(price=400, hot=false))
Cheese(Topping(below=Puffy))
Pattern(pattern=a*b, flags=0, compiled=true, normalizedPattern=a*b, ...
ReflectiveToString()
```

8.3.5 Reflektive Methodenaufrufe

Aufruf von `Method`-Objekten

Objekte der Klasse `Method` repräsentieren Methoden. Die wichtigste „Operation“ von Methoden ist der Aufruf. Die Methode `invoke` macht genau das:

```
Object  invoke(Object x, Object... args)
```

`invoke` richtet einen Aufruf der repräsentierten Methode mit den Argumenten `args` an das Objekt `x` und gibt das Ergebnis als `Object` zurück.

Bei statischen Methoden wird `x` ignoriert und kann auch `null` sein. `void`-Methoden liefern entsprechend `null` zurück.

Auswahl einer `String`-Methode gemäß Kommandozeile

Das folgende Programm ruft eine `String`-Methode auf, deren Parameterliste nur aus Strings besteht. Es erwartet auf der Kommandozeile nacheinander

1. einen String, an den der Methodenaufruf geht,
2. den Namen einer `String`-Methode, deren Argumente Strings sind und
3. die String-Argumente für den Methodenaufruf.

```
001 import java.lang.reflect.*;
002 import java.util.*;
003
004 public class InvokeStringMethod {
005     public static void main(String... args) throws ReflectiveOperationException {
006         String target = args[0];
007         String methodName = args[1];
008         Class<?>[] paramTypes = new Class<?>[args.length - 2];
009         Arrays.fill(paramTypes, String.class);
010         Method method = String.class.getDeclaredMethod(methodName, paramTypes);
011         Object[] restArgs = Arrays.copyOfRange(args, 2, args.length);
012         Object result = method.invoke(target, restArgs);
013         System.out.println(result);
014     }
015 }
```

Listing 8.17: Aufruf einer beliebigen String-Methode, die Strings akzeptiert, via Reflection.

Das Programm geht folgendermaßen vor:

- 006: Das Zielobjekt `target` des Methodenaufrufs ist das erste Kommandozeilenargument.
- 007: Das zweite Kommandozeilenargument legt den Methodennamen `methodName` fest.
- 008–009: Das Array `paramTypes` wird mit so vielen `String`-Typobjekten gefüllt, wie String-Argumente nach dem Methodennamen folgen.
- 010: Die gesuchte Methode ergibt sich aus dem Namen und den Parametertypen.
- 011: Die restlichen Kommandozeilenargumente nach dem Methodennamen werden in ein neues Array kopiert, das die Argumente des Methodenaufrufs enthält.
- 012: Das ist die entscheidende Zeile: `invoke` ruft die Methode `method` mit den Argumenten `restArgs` auf, `result` hält den Rückgabewert fest.
- 013: Zur Kontrolle wird das Ergebnis ausgegeben.

Hier sind einige Programmstarts zu sehen:

```
$ java InvokeStringMethod simsalabim length
10
```

```
$ java InvokeStringMethod simsalabim compareTo abrakadabra
4
$ java InvokeStringMethod simsalabim replaceFirst m M
siMsalabim
$ java InvokeStringMethod simsalabim matches sim.+bimm
false
$ java InvokeStringMethod simsalabim matches sim.+bim
true
```

8.3.6 Erzeugen neuer Objekte

Neben der Untersuchung und Manipulation bestehender Objekte können auch neue Objekte erzeugt werden. Der einfachste Weg führt über die Methode

`Constructor`-Objekte produzieren neue Objekte

```
T  newInstance()
```

der Klasse `Class<T>`, die weiter vorne (Seite 531) vorgestellt wurde. Diese Methode ruft, sofern verfügbar, den Default-Konstruktor auf.

Die gleich benannte Methode

```
T  newInstance(Object... args)
```

in der Klasse `Constructor<T>` (Seite 535) entspricht dem Aufruf eines Custom-Konstruktors. Für sie gelten die gleichen Randbedingungen wie für die `Class`-Methode.

Anlegen von Arrays mit Reflection

Schließlich definiert die Klasse `Array` die beiden statischen Methoden

```
static Object  newInstance(Class<?> type, int length)
static Object  newInstance(Class<?> type, int... lengths)
```

zum Anlegen eines neuen eindimensionalen beziehungsweise mehrdimensionalen Arrays mit dem Elementtyp `type`.

Die letzten beiden Methoden erleichtern den Umgang von generischen Methoden und Klassen mit Arrays, der ansonsten etwas schwierig ist. Mit Kenntnis des gewünschten Elementtyps kann eine generische Methode zur Laufzeit ein passendes Array allozieren. Die folgende Methode erzeugt ein Array der Länge `length` und füllt es mit Kopien des Elements `x`:

```
static <T> T[] make(T x, int length) {
    @SuppressWarnings("unchecked")
```

```
    T[] result = (T[])Array.newInstance(x.getClass(), length);
    // wahlweise auch: Arrays.fill(result, x);
    for(int i = 0; i < length; i++)
        result[i] = x;
    return result;
}
```

Listing 8.18: Generische Factory-Methode für ein Array, das mit einem gegebenen Element gefüllt wird.

Compilerwarnung wegen fehlender statischer Typinformation

Ohne Rückgriff auf Reflection ist das nicht möglich, weil der Compiler ansonsten Code für einen Konstruktoraufruf für ein Array erzeugen müsste, dessen Elementtyp `T` er nicht kennt. Der Compiler warnt allerdings beim `newInstance`-Aufruf, weil er alleine mit den statischen Informationen im Quelltext nicht sicherstellen kann, dass der Typecast die gewünschte Wirkung haben wird.

Das folgende Programm überprüft, ob die erzeugten Arrays den korrekten Laufzeittyp haben:

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

public class ArrayMaker {
    public static void main(String... args) {
        String[] array = make(args[0], 3);
        Object[] objects = array;
        objects[0] = "Abrakadabra";
        objects[1] = new Date();
    }

}
```

Listing 8.19: Erzeugen von Arrays mit einer generischen Factory-Methode.

Das Programm stürzt planmäßig mit einem Laufzeitfehler ab, weil `make` ein Array für Strings liefert, in das das Programm ein `Date`-Objekt einzufügen versucht:

```
$ java ArrayMaker foo
Exception in thread "main" ArrayStoreException: java.util.Date
        at ArrayMaker.main(ArrayMaker.java)
```

▶ Während Arrays in Java zur Laufzeit ihren Elementtyp kennen, gilt das nicht für generische Methoden und Klassen, wie zum Beispiel `ArrayList`. Das Array `array` und die Liste `list` in

```
String[] array = new String[5];
List<String> list = new ArrayList<String>();
```

bieten beide Platz für Strings. Bei `array` garantiert die JVM, dass nur Strings eingefügt werden. Bei `list` kann die JVM zu Laufzeit nichts überprüfen, weil sie keine Informationen über den Elementtyp der Liste hat.

Das folgende Programm demonstriert diesen Unterschied. Es benutzt Reflection, um ein Element des *falschen* Typs `Date` zum einen in ein String-Array und zum anderen in eine `String`-Liste zu speichern. Die statische Typprüfung des Compilers ist bei Reflection generell außer Kraft gesetzt, er übersetzt das Programm daher klaglos:

```
001 import java.lang.reflect.*;
002 import java.util.*;
003
004 public class RuntimeType {
005     public static void main(String... args) throws ReflectiveOperationException {
006         if(args.length == 0) {
007             String[] array = new String[5];
008             Array.set(array, 0, new Date()); // IllegalArgumentException
009         }
010         else {
011             List<String> list = new ArrayList<>();
012             Method add = List.class.getMethod("add", Object.class);
013             add.invoke(list, new Date());
014             System.out.println(list);
015         }
016     }
017 }
```

Listing 8.20: Speichern eines falschen Elementes via Reflection in einem Array oder einer Liste.

Der Aufruf ohne Kommandozeilenargumente erzeugt ein String-Array (007) und versucht dann, über Reflection ein `Date`-Objekt darin zu speichern (008). Das Programm bricht mit einer `IllegalArgumentException` ab, weil die JVM über den Elementtyp des Arrays (`String`) Bescheid weiß und das `Date`-Objekt als unzulässig erkennt:

```
$ java RuntimeType
Exception in thread "main" java.lang.IllegalArgumentException:
    array element type mismatch
    at java.lang.reflect.Array.set(Native Method)
```

Beim Aufruf mit beliebigen Kommandozeilenargumenten gelingt es, in die Liste `list`, die für `String`-Elemente definiert ist (011), über einen reflektiven Aufruf von `add` (012) ein `Date`-Objekt einzufügen (013), wie die Ausgabe (014) zeigt:

```
$ java RuntimeType x
[Thu Dec 08 08:09:02 CET 2011]
```

Die JVM zuckt hier nicht mit der Wimper, weil sie nur eine Liste von Objekten beliebiger Typen sieht, in der ein `Date`-Objekt keinen Verdacht erregt.

Selbstverständlich ist ein `Date`-Objekt nie und nimmer ein String. Daher kommt der „Betrug“ bei dem Versuch ans Licht, anschließend aus der vermeintlichen String-Liste tatsächlich einen String zu holen. Fügt man in das Programm noch die Anweisung

```
String string = list.get(0);
```

ein, dann bricht es mit einer `ClassCastException` ab:

```
$ java RuntimeType x
[Thu Dec 08 08:09:05 CET 2011]
Exception in thread "main" java.lang.ClassCastException:
    java.util.Date cannot be cast to java.lang.String
```

An diesem Beispiel wird eine Schattenseite der Reflection sichtbar: Sie umgeht die hochentwickelte und gründliche statische Typprüfung des Compilers. Infolgedessen fliegen Typfehler nicht zur Übersetzungszeit, sondern erst zur Laufzeit auf, wo sie schwerer zu finden und entsprechend kostspieliger zu beheben sind. ◀

Zusammenfassung

- Zu jedem Java-Typ gibt es genau ein **Typobjekt** der Klasse `Class`.
- Die statische `Class`-Methode `forName` liefert das **Typobjekt** zu einem Klassennamen, der als String gegeben ist.
- Aus einem Typobjekt kann mit `newInstance` **ein neues Objekt** erzeugt werden.
- Mit diesen Mitteln können flexible **Factory-Methoden** definiert werden.
- Die Reflection-Klassen `Class`, `Constructor`, `Method`, `Field` und andere repräsentieren die **Bausteine einer Klassendefinition**.
- Die **Bestandteile** einer unbekannten Klasse können mit **Reflection-Methoden durchsucht** werden.
- Weitere Reflection-Methoden **manipulieren die Objektvariablen** eines Objekts. Zugriffsschutz-Modifier spielen dabei keine Rolle.

Aufgaben

Aufgabe 1: Kontrolle der Standardein- und -ausgabe

Manche Programme lesen Eingaben von der Standardeingabe oder schreiben Ergebnisse auf die Standardausgabe. Mit I/O-Redirection (Abschnitt 1.1.2) und Pipes (Anhang B) kann die Standardein- und -ausgabe umgelenkt werden, aber das geschieht außerhalb von Java auf der Ebene des Betriebssystems.[17] Schreiben Sie eine Klasse `CaptureStandardIO`, die das effizienter innerhalb einer JVM abwickelt. `CaptureStandardIO` definiert die folgenden Methoden:

`CaptureStandardIO(String classname)`
: Konstruktor mit dem qualifizierten Namen einer Klasse, die eine `main`-Methode enthält.

`String callMain(String input, String... args)`
: Ruft die `main`-Methode der im Konstruktor angegebenen Klasse mit den Kommandozeilenargumenten `args` auf. Füttert den String `input` als Standardeingabe und liefert einen String mit der gesamten Standardausgabe als Ergebnis zurück. Diese Methode geht davon aus, dass das aufgerufene Programm Text, das heißt keine Binärdaten, einliest und ausgibt.

`byte[] callMain(byte[] input, String... args)`
: Wie die vorhergehende Methode, versorgt die Standardeingabe aber mit dem Inhalt des Byte-Arrays `input` und liefert ein Byte-Array mit der Standardausgabe zurück.

`void callMain(Path input, Path output, String... args)`
: Wie die vorhergehende Methode, versorgt aber die Standardeingabe mit dem Inhalt der Datei `input` und überträgt die Standardausgabe in die Datei `output`.

Verwenden Sie zur Lösung die folgenden `System`-Methoden, die die Standardein- und -ausgabe des laufenden Programms an neue Streams knüpfen:

```
static void  setIn(InputStream in)
static void  setOut(PrintStream out)
static void  setErr(PrintStream err)
```

[17] Dabei sind verhältnismäßig teure Betriebssystem-Prozesse im Spiel.

Das folgende Programm erwartet eine Grußformel auf der Kommandozeile und liest einen Namen von der Standardeingabe. Es kombiniert daraus einen Gruß, den es auf die Standardausgabe schreibt:

```
import java.io.*;

public class FriendlyGreeting {
    public static void main(String[] args) throws IOException {
        String greeting = args[0];
        String whom = "";
        int code = System.in.read();
        while(code >= 0) {
            whom += (char)code;
            code = System.in.read();
        }
        System.out.println(greeting + ", " + whom.trim() + "!");
    }
}
```

Listing 8.21: Programm, das einen Gruß von der Kommandozeile und einen Gegrüßten von der Standardeingabe liest.

Der folgende Aufruf demonstriert die Arbeitsweise (Benutzereingaben unterstrichen):

```
$ java FriendlyGreeting Hello
World
Hello, World!
```

Die folgende `main`-Methode steuert `FriendlyGreeting` mithilfe von `CaptureStandardIO` fern:

```
public class CaptureStandardIOMain {
    public static void main(String... ignored) throws Exception {
        String[] args = new String[] {"At your command"};
        String input = "Master";
        CaptureStandardIO standardIO = new CaptureStandardIO("FriendlyGreeting");
        String output = standardIO.callMain(input, args);
        System.out.println("program says: " + output);
    }
}
```

Listing 8.22: Test der Fernsteuerung einer Konsolenanwendung.

Das Ergebnis ist:

```
$ java CaptureStandardIOMain
program says: At your command, Master!
```

Aufgabe 2: Automatisches Pickup

Reflection ist der Schlüssel, um Code von statisch fixierten konkreten Klassennamen zu entkoppeln. Stattdessen werden zur Laufzeit aus Strings Typobjekte ermittelt (`forName`) und daraus Objekte erzeugt (`newInstance` und seine Verwandtschaft). In den vorhergehenden Beispielen dieses Kapitels gibt der Anwender die konkreten Klassennamen vor, indem er sie beispielsweise auf der Kommandozeile nennt, wie etwa in `PizzaDeluxeMain` (Listing 8.6). Eine andere Möglichkeit ist es, ein Programm selbst im Filesystem nach passenden Bytecode-Dateien suchen zu lassen.

Als Beispiel dient ein simples Zahlenraten-Spiel. Das Spielprogramm denkt sich eine geheime zweistellige Zahl aus, die die beiden Spieler zu erraten versuchen. Sie kommen abwechselnd an die Reihe. Nach jedem Rateversuch antwortet das Spielprogramm mit „High“ oder „Low“, wenn der Rateversuch zu hoch oder zu tief lag. Sobald ein Spieler die Zahl errät, hat er gewonnen. Nach hundert Runden ohne Gewinner gilt ein Spiel als unentschieden.

Informatiker spielen allerdings nicht selbst, sondern entwickeln Klassen, die spielen. Zwei solche Klassen treten gegeneinander an und versuchen im Namen ihres jeweiligen Schöpfers zu gewinnen. Spieler-Klassen implementieren das folgende Interface:

```
import java.util.*;

public interface HighLowPlayer {
    int guess(Map<Integer, Boolean> history);
}
```

Listing 8.23: Interface für alle Spieler-Klassen.

Das Spielprogramm ruft die Methode `guess` auf und übergibt ihr eine Map mit allen bisherigen Rateversuchen nebst Ergebnissen. Ein Ergebnis ist `true`, wenn der Versuch zu hoch war, und `false` ansonsten. Der Rückgabewert der Methode ist der nächste Rateversuch des Spielers. Hier zwei nicht sonderlich intelligente Spieler-Klassen:[18]

```
import java.util.*;

public class HLPlayer50 implements HighLowPlayer {
```

[18] Ein intelligenter Spieler ist nicht so leicht zu implementieren. Geradlinige binäre Suche ist gegen Ende etwas riskant, wenn man dem Gegner nicht die entscheidende Information zum Gewinnen liefern will. Andererseits endet das Spiel sicher nach einhundert Runden ...

```
    public int guess(Map<Integer, Boolean> history) {
        return 50;
    }
}
```

Listing 8.24: Zahlenratenspieler, der davon überzeugt ist, dass 50 stimmt.

```
import java.util.*;

public class HLPlayerIncrement implements HighLowPlayer {
    private int next = 10;

    public int guess(Map<Integer, Boolean> history) {
        return next++;
    }
}
```

Listing 8.25: Zahlenratenspieler, der nacheinander alle Zahlen ausprobiert.

Das Spielprogramm erwartet die Klassennamen von zwei Spielern auf der Kommandozeile und lässt diese beiden gegeneinander antreten:

```
import java.io.*;
import java.nio.file.*;
import java.util.*;

public class HighLowGame {
    public static void main(String... args) throws IOException, ReflectiveOperationExcept
        final List<HighLowPlayer> players = new ArrayList<>();
        // Ersetzen Sie die nächsten beiden Zeilen
        players.add((HighLowPlayer)Class.forName(args[0]).newInstance());
        players.add((HighLowPlayer)Class.forName(args[1]).newInstance());
        Collections.shuffle(players);
        final Map<Integer, Boolean> history = new HashMap<>();
        HighLowPlayer winner = null;
        int round = 0;
        final int secret = new Random().nextInt(90) + 10;
        while(winner == null && round < 100) {
            final HighLowPlayer player = players.get(round%2);
            final int guess = player.guess(history);
            if(guess == secret)
                winner = player;
            else
                history.put(guess, guess > secret);
            round++;
        }
        System.out.printf("winner: %s%n", winner == null? "none": winner.getClass().getSi
    }
}
```

Listing 8.26: Zahlenraten-Spielprogramm.

Das Programm beschränkt seine Ausgaben auf das Wesentliche:

```
$ java HighLowGame HLPlayerIncrement HLPlayer50
winner: HLPlayerIncrement
$ java HighLowGame HLPlayer50 HLPlayer50
winner: none
```

Ändern Sie das Programm ab, sodass es selbst im Filesystem nach Bytecode-Dateien mit Spieler-Klassen sucht. Die ersten beiden Klassen, die es findet, treten gegeneinander an. Ihr Programm akzeptiert einen einzigen Kommandozeilenparameter mit dem Directory, in dem gesucht wird. Jetzt arbeitet das Programm folgendermaßen:

```
$ java HighLowGame .
picked up player: HLPlayer50
picked up player: HLPlayerIncrement
winner: HLPlayerIncrement
```

Eine neue Spieler-Klasse kann heißen, wie sie will. Sie muss nur übersetzt und der Bytecode in einem passenden Directory deponiert werden. Er wird dann automatisch „aufgesammelt“.

Erweitern Sie `HighLowGame` so, dass es einen Directorybaum rekursiv nach Bytecode von Spielern durchsucht und die Namen der gefundenen Klassen in einer numerierten Liste präsentiert. Die Eingabe von zwei Nummern lässt die betreffenden Spieler gegeneinander antreten.

Aufgabe 3: Mock-Objekte

Zum Testen sind oft Klassen nützlich, die ein gegebenes Interface implementieren, ohne bereits die volle Funktionalität zu bieten. Dabei reicht es aus, wenn der Compiler die Klassen vorläufig akzeptiert, auch wenn es sich bloß um „Potemkinsche Dörfer“ handelt. Solche Klassen bezeichnet man auch als **Mock-Klassen**, ihre Objekte als Mock-Objekte.

Schreiben Sie eine Anwendung `MockMaker`, die auf der Kommandozeile den qualifizierten Klassennamen eines Interface erwartet und den Quelltext einer Mock-Klasse in eine Quelltextdatei im aktuellen Directory schreibt. Die Mock-Klasse soll den Namen des Interface mit dem Präfix „Mock“ haben.

Das folgende Beispiel zeigt die Arbeitsweise:

```
$ java MockMaker java.lang.Runnable
$ cat MockRunnable.java
public class MockRunnable implements java.lang.Runnable {
        public void run() {
                return;
        }
}
$ javac MockRunnable.java
$
```

Ignorieren Sie zur Vereinfachung generische Interfaces. Achten Sie aber darauf, dass Interfaces abgeleitet sein können und dann Methoden erben, die Mock-Klassen ebenfalls implementieren müssen. Alle Methoden, die ein Ergebnis liefern, geben den Defaultwert des betreffenden Typs zurück. (`false` für `boolean`, `'\0'` für `char`, `0.0` für Floatingpoint-Typen, `0` für ganzzahlige Typen und `null` für Referenztypen.)

Erweitern Sie Ihre Lösung so, dass die Methoden der generierten Mock-Klassen ihre Aufrufe mit den Argumentwerten auf der Standardausgabe protokollieren. Der folgende Aufruf zeigt den Anfang einer (längeren) Mock-Klasse:

```
$ java MockMaker org.w3c.dom.Document
$ head MockDocument.java
public class MockDocument implements org.w3c.dom.Document {
        public org.w3c.dom.Node adoptNode(org.w3c.dom.Node p0) {
                System.out.printf("adoptNode(%s)%n", p0.toString());
                return null;
        }
        public org.w3c.dom.Attr createAttribute(java.lang.String p0) {
                System.out.printf("createAttribute(%s)%n", p0.toString());
                return null;
        }
        public org.w3c.dom.Attr createAttributeNS(java.lang.String p0, java.lang.String p
        ...
```

Das gleiche Problem lässt sich auch mit Annotationen (siehe Seite 605) lösen.

Aufgabe 4: UMLet-Dateien

Mit dem Open-Source-Programm UMLet[19] können einfache, statische UML-Klassendiagramme gezeichnet werden. UMLet speichert diese Diagramme in XML-Dokumenten mit der Extension `uxf` ab, deren Aufbau unschwer zu durchschauen ist. Der Inhalt von Boxen ist dabei komplett in Textknoten enthalten, deren Syntax mit der Ausgabe des Beispielprogramms `UMLClassBox` (Listing 8.10) übereinstimmt.

[19] Quelltext und übersetzte Fassungen sind verfügbar auf `http://code.google.com/p/umlet`.

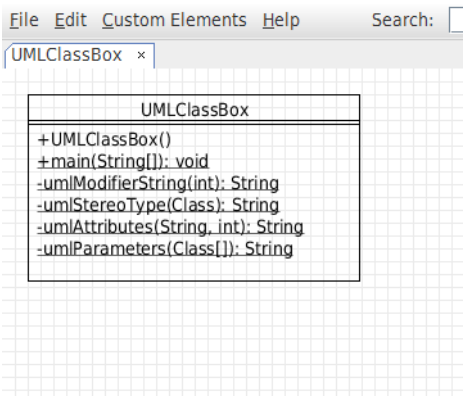
Erweitern Sie `UMLClassBox` zu einem neuen Programm `UMLetUXF`. Dieses Programm erzeugt aus Bytecode-Dateien eine `uxf`-Datei, die in UMLet geladen und dort weiter bearbeitet werden kann.

`UMLetUXF` übergeht die Beziehungen zwischen Typen und produziert nur isolierte Boxen, die später mit UMLet verknüpft werden können.

Konstruieren Sie die XML-Struktur im Programm zunächst als DOM (Seite 211) und geben Sie dieses am Ende aus.

Eine kleine Hürde sind die in `uxf`-Dateien verwendeten Pixelangaben für Positionen und Größen der Diagrammboxen. Hier reicht eine grobe Abschätzung aus, die sich aus dem Text (Zeilenanzahl, längste Zeile) in den Boxen ergibt.[20]

Ein so erzeugtes Klassendiagramm von `UMLClassBox` (Listing 8.10) stellt UMLet etwa folgendermaßen dar:



Produzieren Sie mit `UMLetUXF` Klassendiagramme von Bibliotheksklassen, wie beispielsweise `String`, `Math` und `Object`.

[20] Beginnen Sie bei einer Schriftgröße von 14 Punkt mit einer Zeilenhöhe von 18 Pixel und einen Spaltenbreite von 7 Pixel, zuzüglich eines festen Aufschlags von 20 Pixel auf Höhe und Breite. Die Schriftgröße lässt sich im Options-Dialog von UMLet einstellen.

Kapitel

9 Annotationen

Lernziele

In diesem Kapitel lernen Sie

- welchen Zweck **Annotationen** in Java haben und wie sie verwendet werden.
- welche **vordefinierten Annotationen** Ihnen bereits zur Verfügung stehen.
- wie Sie selbst **neue Annotationen** definieren können.
- wie Annotationen mithilfe von Reflection (Kapitel 8) **zur Laufzeit** gefunden und ausgewertet werden können.
- wie Sie **Annotationprozessoren** schreiben, die während der Übersetzung ablaufen und dabei *innerhalb des Compilers* aus Annotationen automatisch neuen Code generieren.

In Kommentaren halten Entwickler Informationen über Einzelteile von Programmen fest, die dem Quelltext alleine nicht oder nur mit Aufwand zu entnehmen sind. Dass Kommentare wichtig sind, zeigen zahlreiche Programmierrichtlinien. Kommentare sind allerdings freier Text und folgen keinem bestimmten Aufbau. Sie eignen sich daher gut für menschliche Leser, aber weit weniger zur automatischen Verarbeitung.

Eine Zwitterrolle spielen Javadoc-Kommentare. Einerseits sind es Kommentare, andererseits enthalten sie syntaktisch fixierte „Tags“, die das Werkzeug Javadoc aufspüren und auswerten kann.

Annotationen sind eine Art konsequenter Fortsetzung von Javadoc-Tags: Sie steuern „Anmerkungen“ zum Quelltext bei, folgen aber einer festen Schreibweise und dürfen nur an bestimmten Stellen platziert werden. Wegen ihres formalen Aufbaus können sie mit geeigneten Werkzeugen automatisch verarbeitet werden. Im Gegensatz zu Javadoc-Kommentaren sind Annotationen aber nicht fixiert, sondern können nach Bedarf neu definiert werden.

Aus programmiersprachlicher Sicht zählen Annotationen zu den Typen von Java. Sie sind eine besondere Art von Interfaces, sowohl hinsichtlich der Definition als auch der Verwendung.

9.1 Idee

Kommentare mit definiertem Aufbau

Annotationen spielen eine ähnliche Rolle wie Kommentare, haben allerdings eine Reihe von Vorteilen:

- Sie unterliegen einer festen Syntax und können beispielsweise nicht versehentlich falsch geschrieben werden.
- Annotationen können wahlweise in den Bytecode übertragen und zur Laufzeit wieder aufgespürt werden. Kommentare überleben den Compiler nicht.
- Die formale Definition erlaubt benannte Argumente mit unterschiedlichen Typen.
- Sowohl der Compiler als auch die JVM bieten Schnittstellen zum Umgang mit Annotationen an.

Javadoc-Tags als Vorläufer

Im Grunde kann man Javadoc als frühen Vorläufer von Annotationen sehen. Allerdings dient Javadoc nur einem einzigen Zweck (eben der Dokumentation), während Annotationen beliebigen Werkzeugen zur Verfügung stehen.[1]

Definition, Anwendung und Auswertung

Bei der Verarbeitung von Annotationen spielen einige Mechanismen zusammen:

- Zunächst legt eine Definition die Eigenschaften von Annotationen fest, wie zum Beispiel den Namen. Dieser Punkt wird für den Augenblick zurückgestellt und in Abschnitt 587 aufgegriffen.
- Der Quelltext verwendet Annotationen, um bestimmte Programmelemente damit zu markieren.
- Der Compiler stellt Schnittstellen zur Verfügung, um Annotationen im Quelltext während der Übersetzung auszuwerten.
- Der Compiler überträgt außerdem unter bestimmten Voraussetzungen Annotationen in Bytecode.
- Getrennte Werkzeuge extrahieren Annotationen aus Bytecode-Dateien und verfahren damit nach eigenem Ermessen.
- Ein Programm kann Annotationen mit Reflection zur Laufzeit aufspüren und und darauf reagieren.

[1] Aus heutiger Sicht könnten Annotationen sogar manche Schlüsselwörter ersetzen, wie beispielsweise die Modifier `volatile` (Seite 422) und `transient` (Seite 140). Das würde die *Sprache* Java vereinfachen, weil Annotationen in Bibliotheken definiert sind und nicht zur Syntax zählen.

Eine Annotation hat allgemein die syntaktische Form

Syntaktisches Merkmal `@`-Zeichen

```
@name(attribute=value, attribute=value, ...)
```

Das führende `@`-Zeichen unterscheidet eine Annotation von anderen Sprachelementen.[2] Der Name einer Annotation (*name*) legt die zulässigen Attribute (*attribute*) und die jeweils erlaubten Werte (*value*) fest. Die Attribute einer Annotation sind eindeutig und kommen nicht mehrfach vor. Ihre Reihenfolge ist ohne Bedeutung. Wenn es nur ein einziges Attribut gibt und wenn dieses eine Attribut außerdem `value` heißt, dann kann die Annotation kürzer geschrieben werden als

Kurzform für einzelnes Attribut names `value`

```
@name(value)
```

Die beiden folgenden Schreibweisen sind also gleichwertig:

```
@SuppressWarnings(value="unchecked")
@SuppressWarnings("unchecked")
```

Bei einer Annotation ohne Attribute können die (leeren) runden Klammern wegfallen. Die zwei nächsten Annotationen sind äquivalent:

```
@Deprecated()
@Deprecated
```

Annotationen sind vor Definitionen erlaubt. Annotiert werden können unter anderem:

Annotation für alle Arten von Definitionen

- Klassen, Aufzählungstypen, Interfaces,
- Konstruktoren,
- Methoden,
- Objekt- und Klassenvariablen, Aufzählungswerte,
- Parameter und
- lokale Variablen.

Syntaktisch stehen Annotationen auf der gleichen Ebene wie Modifier. Per Konvention sollten Annotationen vor anderen Modifiern genannt werden, aber der Compiler erzwingt das nicht.

Annotationen gleichrangig mit Modifiern

Für jede einzelne Annotation ist geregelt, auf *welche* Arten von Definitionen sie an-

Zulässig vor bestimmten Definitionen

[2] Formal ist `@` ein selbstständiges syntaktisches Element und kann beispielsweise mit Zwischenraum vom Namen abgesetzt werden. Das ist allerdings unüblich.

wendbar ist. Beispielsweise ist `@SuppressWarnings` vor *allen* Definitionen zulässig, `@Deprecated` dagegen nicht vor lokalen Variablen. Das folgende Beispiel markiert eine Methode, die möglichst nicht mehr verwendet werden soll:

```
@Deprecated public void oldStyle() {...}
```

Im nächsten Beispiel soll der Compiler keine Warnungen wegen potenzieller Typfehler ausgeben, siehe `ArrayMaker`:

```
@SuppressWarnings("unchecked") public void makeArray() {...}
```

Lebensdauer von Annotationen

Eine weitere wichtige Eigenschaft von Annotationen ist ihre Lebensdauer (*retention*). Dabei sind die folgenden Abstufungen möglich:

Quelltext
: Erhalt bis zum Compiler, der sie entfernt. Diese Stufe entspricht Kommentaren, die der Compiler ebenfalls entfernt.

Bytecode
: Erhalt im Bytecode bis zum Laden in die JVM. Externe Werkzeuge, die den Bytecode analysieren, sehen diese Annotationen. Im laufenden Programm sind sie dagegen verschwunden.

Laufzeit
: Überleben bis in das laufende Programm. Annotationen mit dieser Lebensdauer stehen dem Programm zur Selbstuntersuchung durch Reflection zur Verfügung.

Beispielsweise hat `@Deprecated` die Retention „Laufzeit“. Diese Annotation ist damit Bestandteil von `class`-Dateien und auch noch im laufenden Programm vorhanden.

Dagegen hat `@SuppressWarnings` nur die Retention „Quelltext“. Nachdem der Compiler seine Bedenken bezüglich des betreffenden fragwürdigen Konstrukts für sich behalten und den Code folgsam übersetzt hat, spielt diese Annotation keine Rolle mehr.

9.2 Vordefinierte Annotationen

Annotationen der Laufzeitbibliothek

Java bringt einige vordefinierte Annotationen mit. Sie werden beispielsweise in der Laufzeitbibliothek selbst verwendet und stehen auch für neuen Code zur Verfügung:

`@Deprecated`

Markiert eine Definition, die man nicht (mehr) verwenden sollte. Über die Gründe sagt diese Annotation nichts aus. In der Regel betrifft sie Methoden und Konstruktoren, gelegentlich auch ganze Klassen, für die es bessere Alternativen gibt. Ein Beispiel ist die `Thread`-Methode `stop()`. Auch die folgende Klasse ist mit `@Deprecated` annotiert:

Abgekündigte Klassen und Methoden

```
@Deprecated public class ProbablyBroken {
}
```

Listing 9.1: Klasse, die nicht mehr verwendet werden soll.

Diese Annotation richtet sich an Nutzer der betreffenden Definition, wie zum Beispiel:

```
public class DeprecatedUsage {
    public static void main(String... args) {
        new ProbablyBroken();
    }
}
```

Listing 9.2: Erzeugt ein Objekt einer abgekündigten Klasse.

Der Java-Compiler gibt eine Warnung aus:

```
$ javac DeprecatedUsage.java
Note: DeprecatedUsage.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

Ein neuer Aufruf mit dem zusätzlichen Schalter `-Xlint` liefert weitere Einzelheiten:

```
$ javac -Xlint DeprecatedUsage.java
DeprecatedUsage.java:
warning: [deprecation] ProbablyBroken has been deprecated
                new ProbablyBroken();
                    ^
```

Die Annotation `@Deprecated` hat den gleichen Sinn wie das Javadoc-Tag `@deprecated`.[3] An diesem Beispiel zeigt sich der Unterschied zwischen einem Javadoc-Tag und einer Annotation:

Javadoc-Tag vs. Annotation

- Das Tag gibt einem (wahrscheinlich menschlichen) *Leser* der Dokumentation einen Hinweis, aber nicht dem Compiler.

[3] Javadoc akzeptiert statt des Javadoc-Tags `@deprecated` auch die Annotation `@Deprecated` und fügt einen entsprechenden Hinweis in die generierte Dokumentation ein.

- Die Annotation macht die Information dagegen auch dem *Compiler* und anderen Werkzeugen zugänglich.

`@Override`

Markierung redefinierender Methoden

Diese Annotation gilt auf Quelltextebene und ausschließlich für Methoden. Sie markiert die Redefinition einer ererbten Methode oder die Implementierung einer abstrakten oder Interface-Methode. Compiler *müssen* eine überzählige `@Override`-Annotation an der falschen Stelle als Fehler ablehnen. Für eine fehlende `@Override`-Annotation gilt das nicht.
Die folgende Basisklasse definiert zwei Methoden:

```
public class Foo {
    public void foo() {
    }

    public void bar() {
    }
}
```

Listing 9.3: Basisklasse zum Ableiten und Testen von `Override`-Annotationen.

Die nächste Klasse redefiniert `foo` korrekt mit einer `@Override`-Annotation. Bei `bar` fehlt die eigentlich nötige Annotation, aber das ist *kein* Fehler. Fehlerhaft ist dagegen die überzählige Annotation bei `baz`, weil `baz` keine Methode *re*definiert:

```
public abstract class SonOfFoo extends Foo {
    @Override public void foo() {
    }

    public void bar() {
    }

    @Override public void baz() {
    }
}
```

Listing 9.4: Korrekte, fehlende und falsche `Override`-Annotationen.

Der Compiler übersetzt die Klasse nicht:

```
$ javac SonOfFoo.java
SonOfFoo.java: error:
method does not override or implement a method from a supertype
        @Override public void baz()
        ^
```

`@Override` hilft Fehler zu vermeiden. Die Annotation macht Methoden sofort sichtbar, die entgegen der Absicht eine Basisklassenmethode *nicht*

redefinieren. Ein typischer Kandidat ist eine `equals`-Methode mit einem anderen Parametertyp als `Object`, die die `Object`-Methode überlädt statt redefiniert.

`@SafeVarargs`

Diese Annotation beseitigt eine Compilerwarnung im Zusammenhang mit Vararg-Parametern eines generischen Typs. Das Problem ist etwas subtiler und wird auf Seite 576 genauer erklärt.

`@SuppressWarnings`

Steuerung der Warnungen des Compilers

Diese Annotation unterdrückt Gruppen von Warnungen des Compilers. Der Attributwert der Annotation legt fest, welche Gruppe die Ausnahme betrifft. Es gibt zwei Schreibweisen für eine und mehrere Gruppen:

```
@SuppressWarnings("group")
@SuppressWarnings({"group", "group", ...})
```

Gruppen sind mit Strings benannt und werden auf Seite 574 erklärt. Die Annotation bezieht sich nur die jeweils annotierte Definition selbst, aber nicht auf den Rest des Quelltexts. Zur Sicherheit sollte man die Annotation nur vor die direkt betroffene Definition stellen und nicht auf unnötig hoher Ebene platzieren.
Beispielsweise steht die Annotation `@SuppressWarnings("unchecked")` in `ArrayMaker` vor der Definition der lokalen Variablen `result`. Man könnte auch die Methode `make` oder gleich die ganze Klasse `ArrayMaker` markieren. Dann wären aber auch Warnungen bezüglich anderer Definitionen unterdrückt, bei denen diese Warnungen vielleicht durchaus angebracht sind.

Nur Definitionen annotierbar

In Java bis Version 7 können Annotationen nur vor Definitionen stehen, aber nicht vor anderen Konstrukten, obwohl das bei `@SuppressWarnings` manchmal nicht passt. Die folgende Variante der Methode `make` von `ArrayMaker` definiert die lokale Variable `result` und weist ihr in der nächsten Anweisung einen Wert zu:

```
static <T> T[] make(T x, int length) {
    T[] result;
    @SuppressWarnings("unchecked")  // hier keine Annotation erlaubt
    result = (T[])Array.newInstance(x.getClass(), length);
    for(int i = 0; i < length; i++)
        result[i] = x;
    return result;
}
```

Listing 9.5: Erzeugen von Arrays mit einer generischen Factory-Methode.

Kritisch ist dabei die Wertzuweisung, die eigentlich annotiert werden müsste. Trotzdem darf die Annotation dort nicht stehen, weil die Wertzuweisung nichts definiert. Der Code wird also so nicht übersetzt. Die

Annotation muss bei der umfassenden Definition der Methode `make` platziert werden, obwohl sie damit eigentlich einen zu großen Einflussbereich erlangt.

Annotationen für Annotationen

Über die hier gezeigten vordefinierten Annotationen hinaus gibt es einige **Meta-Annotationen**, um neue, selbst definierte Annotationen zu markieren. Diese Meta-Annotationen werden im Abschnitt 9.3 vorgestellt.

9.2.1 Warnungen des Java-Compilers

Compilerwarnungen für suspekte Konstrukte

Der Java-Compiler produziert eine Anzahl Warnungen, die auf erfahrungsgemäß heikle Konstrukte und Situationen hinweisen, in denen der Compiler die Korrektheit seiner Annahmen nicht mehr garantieren kann. Die konkrete Auswahl und die Umsetzung von Warnungen hängt vom einzelnen Compilerhersteller ab.

Warnungen des Oracle-Java-Compilers

Der Compiler des Oracle-JDK kennt verschiedene Gruppen, die der folgende Aufruf mit dem Schalter `-X` auflistet: [4]

```
$ javac -X
...
-Xlint:{all,cast,classfile,deprecation,dep-ann,divzero,empty,fallthrough,finally,
options,overrides,path,processing,rawtypes,serial,static,try,unchecked,varargs,
-cast,-classfile,-deprecation,-dep-ann,-divzero,-empty,-fallthrough,-finally,
-options,-overrides,-path,-processing,-rawtypes,-serial,-static,-try,-unchecked,
-varargs,none} Enable or disable specific warnings
```

Alle Warnungen lassen sich komplett aktivieren und stilllegen:[5]

```
javac -Xlint:all ...
javac -Xlint:none ...
```

Aktivieren und Stilllegen von Gruppen

Die Schalter

```
-Xlint:group
-Xlint:-group
```

[4] Alle Compilerschalter, die mit `-X` beginnen, sind herstellerabhängig.

[5] Der Schalter `-Xlint` aktiviert die *empfohlenen* Warnungen. Derzeit werden aber alle Warnungen „empfohlen“, deshalb haben `-Xlint` und `-Xlint:all` die gleiche Wirkung. Das muss nicht so bleiben und kann von von verschiedenen Compilerherstellern unterschiedlich geregelt werden.

betreffen eine einzelne Gruppe und berühren die anderen nicht. Mit einer durch Kommata getrennten Liste[6] lassen sich verschiedene Gruppen auswählen oder ausblenden. Beispielsweise aktivieren die gleichwertigen Schalter

```
-Xlint:all,-divzero
-Xlint:-divzero,all
-Xlint:all -Xlint:-divzero
-Xlint -Xlint:-divzero
```

alle Warnungen (`all`) außer „Division durch null“ (`-divzero`).

Beispiel: Umgang mit Division durch null

Das folgende Programm weckt das Misstrauen des Compilers:

```
import java.util.*;

public class CompilerWarnings {
    public static void main(String... args) {
        int x = 2/0;
        List<String> list = new ArrayList();
    }
}
```

Listing 9.6: Programm mit Compilerwarnungen.

Beim Übersetzen ohne Schalter äußert der Compiler nur einen Hinweis („Note“), aber noch keine Warnung:

```
$ javac CompilerWarnings.java
Note: CompilerWarnings.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Ein neuer Aufruf mit dem Schalter `-Xlint:unchecked` aktiviert die Warnungen der Gruppe `unchecked`:

```
$ javac -Xlint:unchecked CompilerWarnings.java
CompilerWarnings.java: warning: [unchecked] unchecked conversion
               List<String> list = new ArrayList();
                                   ^
  required: List<String>
  found:    ArrayList
1 warning
```

6 `-Xlint` kann auch mehrmals hintereinander angegeben werden.

Die Warnungen aller Gruppen macht der Schalter `-Xlint:all` sichtbar. Die zwei neuen Warnungen beziehen sich auf Division durch null und auf die Verwendung der generischen Klasse `ArrayList` ohne Typargument:

```
$ javac -Xlint:all CompilerWarnings.java
CompilerWarnings.java: warning: [divzero] division by zero
                int x = 2/0;
                        ^
CompilerWarnings.java: warning: [rawtypes] found raw type: ArrayList
                List<String> list = new ArrayList();
                                    ^
  missing type arguments for generic class ArrayList<E>
  where E is a type-variable:
    E extends Object declared in class ArrayList
CompilerWarnings.java: warning: [unchecked] unchecked conversion
                List<String> list = new ArrayList();
                                    ^
  required: List<String>
  found:    ArrayList
3 warnings
```

Gruppen von Warnungen

Die folgende Liste erklärt die verschiedenen Gruppen:

`cast` Überflüssige Typecasts, wie zum Beispiel in der Anweisung `int i = (int)0;`.

`classfile`
: Verdächte Bytecode-Dateien, beispielsweise mit anderen Merkmalen als die ebenfalls enthaltene Versionsangabe eigentlich gestattet.

`deprecation`
: Verwendung von Klassen oder Methoden, die als `Deprecated` markiert sind.

`dep-ann` Annotation `@Deprecated` und `@deprecated`-Tag im Javadoc-Kommentar widersprechen sich.

`divzero` Potenzielle Division durch null.

`empty` Leere Anweisung nach `if`.

`fallthrough`
: `case`-Block in einer `swicth`-Anweisung ohne abschließendes `break`.

`finally` `return`-Anweisung in einem `finally`-Block. Diese Konstruktion kann eine andere `return`-Anweisung im `try`-Block maskieren und zu schwer nachvollziehbarem Verhalten führen.

`options` Aufruf des Compilers mit einer älteren Version für den Quelltext (Schalter `-source`) als für die Laufzeitbibliothek (`-bootclasspath`). Dabei kann Bytecode entstehen, der auf der älteren Plattform nicht funktioniert, weil er sich auf die neuere Laufzeitbibliothek bezieht.

`overrides`
Fehlende `@Override`-Annotation.

`path` Unerreichbares Element im Classpath.

`processing`
Annotation gefunden, für die sich kein Prozessor zuständig erklärt (siehe Seite 595).

`rawtypes`
Verwendung einer generischen Klasse oder eines generischen Interface ohne Typargument.

`serial` Fehlende oder falsche Definition von `serialVersionUID` in einer Klasse, die das Interface `Serializable` implementiert (siehe Seite 143).

`static` Bezug auf ein statisches Element über eine Objektreferenz statt über den Klassennamen.

`try` Verdächtiger ARM-Block (siehe Seite 36), wie beispielsweise

- ARM-Block mit einer Ressource, die nicht verwendet wird,
- ARM-Block mit einem expliziten `close`-Aufruf,
- Definition einer `close`-Methode mit einer anderen Exception als `IOException`.

`unchecked`
Einsatz eines generischen Typs in einem Kontext, in dem die statische Typprüfung des Compilers nicht mehr sicherstellen kann, dass Typfehler zur Laufzeit ausgeschlossen sind.

`varargs` Vararg-Parameter mit einem generischen Typ, siehe nächster Abschnitt.

▶ Der Java-Compiler hinterlegt in Bytecode-Dateien die Java-Version. Die beiden Bytes an den Offsets 6 und 7 einer `class`-Datei speichern die Neben- und die Hauptversion (Minor- und Major-Version). Es gilt der folgende Zusammenhang:

Java-Version	Major-Version	Minor-Version
1.0	45	0–3
1.1	45	mehrere
1.2	46	0
1.3	47	0
1.4	48	0
5	49	0
6	50	0
7	51	0
8	noch offen	0

Java-Compiler können Quelltext älterer Versionen lesen (Schalter `-source`) und auch Bytecode älterer Versionen erzeugen (Schalter `-target`). Die zulässigen Sprachmittel sind dabei entsprechend der Version eingeschränkt. Beispielsweise wechselt `enum` mit Version 5 seine Rolle von einem regulären Identifier zu einem Schlüsselwort.

Es gibt externe Werkzeuge, sogenannte *Java backporting tools*, die in gewissen Grenzen neuere Bytecode-Dateien in ältere Versionen transformieren können. ◀

9.2.2 Annotation `SafeVarargs`

Arrays und generische Methoden

Dieser Abschnitt hat nur vordergründig mit Annotationen zu tun. Es geht in erster Linie um das etwas schwierige Verhältnis zwischen Arrays und generischen Methoden.

Die Annotation `@SafeVarargs` unterdrückt zwei Warnungen im Zusammenhang mit Vararg-Parametern eines generischen Typs. Allerdings haben diese Warnungen oft einen guten Grund, daher sollte die Annotation mit Bedacht eingesetzt werden.

Harmloses Beispiel

Die folgende Klasse definiert einen statische Methode `asArray` mit einem generischen Vararg-Parameter. `main` ruft diese Methode mit einem einzelnen String auf:

```
public class SafeVarargsAnnotation {
    static <T> T[] asArray(T... data) {
        return data;
    }
    public static void main(String[] args) {
        String[] stringArray = asArray("Hello");
```

```
    }
}
```

Listing 9.7: Unkritischer Aufruf einer generischen Vararg-Methode.

Der Aufruf der Methode in `main` ist unkritisch: Der Compiler erkennt das `String`-Argument, legt ein String-Array an und übergibt dieses an die Methode. Das dabei erzeugte Array nimmt nur Strings auf, wie der folgende Versuch zeigt, der erwartungsgemäß mit einer `ArrayStoreException` endet:

```
String[] stringArray = asArray("Hello");
Object[] objectArray = stringArray;
objectArray[0] = 1;         // ArrayStoreException
```

Listing 9.8: Nachweis, dass `asArray` tatsächlich ein String-Array liefert.

Das Verderben nimmt seinen Lauf, wenn der einfache String durch ein Objekt des generischen Typs `BoxedValue<String>` ersetzt wird, der nur einen Wert kapselt: Bösartiges Beispiel

```
class BoxedValue<T> {
    private final T value;

    public BoxedValue(T value) {
        this.value = value;
    }

    public T unbox() {
        return value;
    }
}
```

Listing 9.9: Unveränderliche generische Klasse, die einen einzelnen Wert kapselt.

Die *Type-Erasure* entfernt das Typargument, sodass das Laufzeitsystem mangels weiterer Informationen nur ein Array für beliebige `BoxedValue`-Elemente erzeugen und an die Methode übergeben kann. Das folgende Programmfragment nutzt diese Lücke aus, um anschließend ein Element eines „falschen“ Typs im Array zu speichern, *ohne dabei eine Exception auszulösen!* Ab diesem Punkt sind alle Annahmen des Compilers über Typen von Objekten hinfällig.

```
BoxedValue<String>[] boxedStringsArray = asArray(new BoxedValue<>("Hello"));
BoxedValue<?>[] boxedObjectsArray = boxedStringsArray;
boxedObjectsArray[0] = new BoxedValue<>(1); // keine Exception!
```

Listing 9.10: Kritischer Aufruf einer generischen Vararg-Methode.

Aus diesem Grund warnt der Compiler völlig zu Recht bei allen

Compilerwarnung auf Verdacht

- Definitionen von Methoden mit generischen Vararg-Parametern:

```
$ javac -Xlint:unchecked SafeVarargsAnnotation.java
SafeVarargsAnnotation.java:
warning: [unchecked] Possible heap pollution from parameterized vararg type T
    static <T> T[] asArray(T... data)
                   ^
```

- Aufrufen solcher Methoden mit Argumenten generischer Typen:

```
$ javac -Xlint:unchecked SafeVarargsAnnotation.java
SafeVarargsAnnotation.java:
warning: [unchecked] unchecked generic array creation for varargs parameter
    BoxedValue<String>[] boxedStringsArray = asArray(new BoxedValue<>("Hello"));
                                             ^
```

Die Wurzel des Übels ist die Rückgabe des Arrays aus `asArray`. Eine Methode mit generischen Vararg-Parametern wäre in Ordnung, wenn man nur sicherstellen könnte, dass unter keinen Umständen ein Element eines falschen Typs in das Array gelangen kann. Für die folgende Methode `printAll`, die ihre Argumente lediglich ausgibt, gilt das beispielsweise. Insbesondere gibt `printAll` das Vararg-Array nicht zurück und schließt damit auch jede spätere Schreiboperation aus.

```
static <T> void printAll(T... data) {
    for(T element: data)
        System.out.println(element);
}
```

Listing 9.11: Generische Vararg-Methode, die das Array nicht ändert.

Compilerwarnung auch in sicheren Fällen

Der Compiler kann das allerdings nicht erkennen und warnt unnötigerweise auch bei der Definition und bei Aufrufen von `printAll`, obwohl das falscher Alarm ist:

```
printAll("Hello");
printAll(new BoxedValue<>("Hello"));
```

Listing 9.12: Aufrufe einer sicheren generischen Vararg-Methode.

Methoden wie `printAll` sind die Adressaten der Annotation `@SafeVarargs`, die die Warnungen bei der Definition und bei Aufrufen stilllegt:

```
@SafeVarargs
static <T> void printAll(T... data) {
    for(T element: data)
        System.out.println(element);
}
```

Listing 9.13: Als harmlos annotierte generische Vararg-Methode.

Begründete Ausnahmen mit Annotation

Mit `@SafeVarargs` muss vorsichtig umgegangen werden. Eine entsprechend annotierte Methode darf das Vararg-Array keinesfalls zurückgeben. Natürlich darf sie selbst, wenn überhaupt, nur Elemente des korrekten Typs (`T`) in das Array schreiben. Schließlich darf sie das Array als Argument nur an andere Methoden übergeben, die selbst vertrauenswürdig mit `@SafeVarargs` annotiert sind.

Die Einhaltung dieser Bedingungen entziehen sich der Kontrolle des Compilers. Der Schalter `-Xlint:varargs` macht entsprechende Konstruktionen sichtbar, unabhängig von der Annotation `@SafeVarargs`. Damit können im Zweifelsfall potenziell problematische Codestellen zuverlässig lokalisiert werden.

9.3 Neue Annotationen

9.3.1 Definition

Neue Art von Referenztypen

Formal sind Annotationen eine weitere Art von Referenztypen, neben Klassen, Interfaces und Aufzählungstypen. Die Definition ähnelt syntaktisch einer Interface-Definition mit dem Schlüsselwort `@interface`:

```
@interface name {
    // body
}
```

Der Rumpf enthält eine beliebige Anzahl von Annotation-Attributen in einer der beiden Formen

```
type name();
type name() default value;
```

Attribute mit Defaultwerten

Das folgende Beispiel definiert eine Annotation namens `Immutable` mit einem `boolean`-Attribut `strong`, das den Defaultwert `true` hat:[7]

[7] Der Name der Annotation legt nahe, dass sie eine unveränderliche Klasse markiert. Bezogen auf Java kann man „starke“ und „schwache“ Unveränderlichkeit unterscheiden: In einer „stark unver-

```
public @interface Immutable {
    boolean strong() default true;
}
```

Listing 9.14: Annotation mit einem `boolean`-Attribut.

Der Compiler übersetzt diese Definition in Bytecode, ebenso wie andere Typdefinitionen. Die Namen von Annotationen konkurrieren mit den anderen Typnamen im selben Package.

Ähnlichkeit mit Methodensignaturen

Annotation-Attribute haben die Gestalt von Methodenköpfen mit einigen Einschränkungen:

- Die Parameterliste ist leer,
- sie haben keine Exceptionsignatur,
- als Ergebnistypen sind nur erlaubt: primitive Typen, `String`, `Class` und abgeleitete generische Typen, Aufzählungstypen, andere Annotation-Typen, eindimensionale Arrays der vorgenannten Typen.

Interface für alle Annotationen

Alle Annotation-Typen implementieren implizit das Interface `Annotation` im Package `java.lang.annotation`. Ein Annotation-Typ kann nur mit `@interface` definiert werden, nicht durch Implementieren des Interface `Annotation`.

9.3.2 Annotation-Werte

Anwendung einer Annotation

Die Anwendung einer Annotation nennt den Namen mit einem `@`-Zeichen, gefolgt von runden Klammern mit einer Liste aller Attribute mit Werten, die der Compiler berechnen kann. Leere runde Klammern können wegfallen. Fehlende Attribute müssen mit einem Defaultwert definiert sein, der in diesem Fall eingesetzt wird.

Die Annotation `Immutable` kann beispielsweise folgendermaßen verwendet werden:

```
@Immutable class Rational {
    private final int num;

    private final int denom;
```

änderlichen“ Klasse sind alle Objektvariablen `final` definiert und selbst „stark unveränderlich“. Eine „schwach unveränderliche“ Klasse kann veränderliche Bestandteile enthalten, stellt aber nach außen keine Möglichkeit zum Ändern des Zustandes zur Verfügung. Eine Aufgabe am Ende des Kapitels greift diese Annotation noch einmal auf.

```
    public Rational(int num, int denom) {
        this.num = num;
        this.denom = denom;
    }

    public int getNum() {
        return num;
    }

    public int getDenom() {
        return denom;
    }
}
```

Listing 9.15: Unveränderliche Klasse.

Alternativ könnte die Annotation auch geschrieben werden als:[8]

```
@Immutable()
@Immutable(strong = true)
@Immutable(strong = 2*3 > 5)
```

Zur Illustration verschiedener Attributtypen sei eine Annotation `AnAnnotation` definiert:[9]

Beispiele verschiedener Attributtypen

```
@interface AnAnnotation {
    int count();
    String greeting();
    Class<? extends Comparable<?>> comparable();
    Color favoriteColor();
    Immutable prop();
    String[] reasons();
}
```

Die folgende Liste zeigt Anwendungen dieser Attribute:

```
@AnAnnotation(count = 10)
@AnAnnotation(greeting = "Hello")
@AnAnnotation(comparable = Boolean.class)
@AnAnnotation(favoriteColor = Color.Blue)
@AnAnnotation(prop = @Immutable(strong = true))
@AnAnnotation(reasons = {"important", "unknown"})
```

Die geschweiften Klammern bei Aufzählungen von Array-Elementen können wegfallen, wenn genau ein Wert genannt ist. Die beiden folgenden Annotationen sind gleichwertig:

---

8 Das letzte (nicht sinnvolle) Beispiel zeigt, dass als Attributwerte Ausdrücke erlaubt sind, die der Compiler berechnen kann.

9 `Color` sei definiert als `enum Color{Red, Green, Blue}`.

```
@AnAnnotation(reasons = {"important"})
@AnAnnotation(reasons = "important")
```

9.3.3 Meta-Annotation `Target`

Auswahl der betroffenen Definitionen

In der Voreinstellung sind Annotationen auf alle Arten von Definitionen anwendbar. In der Regel ist das aber nicht sinnvoll, deshalb lässt sich eine Annotation bei der Definition auf bestimmte Anwendungsfälle einschränken. Zur Wahl steht der folgende Katalog. (Die Werte in der linken Spalte sind Elemente des Aufzählungstyps `ElementType`.)

`TYPE`	Klassen, Aufzählungstypen, Interfaces
`CONSTRUCTOR`	Konstruktoren
`METHOD`	andere Methoden
`FIELD`	Objekt- und Klassenvariablen, Aufzählungswerte
`PARAMETER`	Parameter
`LOCAL_VARIABLE`	lokale Variablen
`ANNOTATION_TYPE`	Definitionen anderer Annotationen

Meta-Annotation betrifft andere Annotationen

Hier geht es um eine Eigenschaft von Annotationen. Diese Eigenschaft wird selbst mit einer Annotation namens `Target` festgelegt. `Target` ist eine Meta-Annotation, weil sie bei der Definition einer anderen Annotation verwendet wird. Sie hat ein einziges Attribut `value` des Aufzählungstyps `ElementType`. Das folgende Beispiel schränkt `Immutable` mit dem Attributwert `ElementType.TYPE` auf Typdefinitionen ein. Vor Methoden, Variablen und so weiter ist `Immutable` damit nicht mehr zulässig.

```
import java.lang.annotation.*;

@Target(ElementType.TYPE)
public @interface Immutable {
    boolean strong() default true;
}
```

Listing 9.16: Annotation eingeschränkt auf Typdefinitionen.

`Target` ist bereits in der Java-Bibliothek vordefiniert. Die Definition könnte dort etwa folgendermaßen aussehen:[10]

[10] Das ist nur eine Skizze, weil eine Annotation nicht auf sich selbst angewendet werden kann.

```
@Target(ElementType.ANNOTATION_TYPE)
@interface Target {
    ElementType[] value() default {ElementType.ANNOTATION_TYPE, ...};
}
```

9.3.4 Meta-Annotation `Retention`

Steuerung der Lebensdauer

Die zweite wichtige Eigenschaft einer Annotation ist ihr Erhalt (*retention*), das heißt, der Horizont ihrer Sichtbarkeit. Es gibt drei Möglichkeiten mit entsprechenden Elementen des Aufzählungstyps `RetentionPolicy`:

`SOURCE` Nur im Compiler sichtbar (wie Kommentare).

`CLASS` Im Compiler sichtbar und als Bytecode-Attribute in `class`-Files eingebettet (Voreinstellung).

`RUNTIME` Im Compiler, im Bytecode und im laufenden Programm via Reflection sichtbar.

Die Meta-Annotation `Retention` mit einem Wert des Aufzählungstyps `RetentionPolicy` definiert den Erhalt einer Annotation. Die folgende Definition von `Immutable` legt die Sichtbarkeit `RetentionPolicy.SOURCE`, das heißt „nur im Compiler“ fest:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface Immutable {
    boolean strong() default true;
}
```

Listing 9.17: Annotation mit Erhalt im Compiler, aber nicht länger.

`@Target` und `@Retention` als Standard-Meta-Annotationen

Die Definition einer neuen Annotation wird in der Regel mit den beiden Meta-Annotationen `@Target` und `@Retention` versehen. Die Voreinstellungen (`@Target` für alle Definitionen, `@Retention` im Bytecode) sind eher selten passend.

▶ Das Werkzeug `javap` (*Java class file disassembler*) macht Annotationen im Bytecode sichtbar. Das folgende Programm definiert drei statisch geschachtelte Annotationtypen (siehe 5.1) mit den drei möglichen Retentions und wendet sie auf drei Klassenvariablen (`a`, `b`, `c`) an.

```
import java.lang.annotation.*;

public class AnnotationLevels {
    @Retention(RetentionPolicy.SOURCE)
    @interface SourceAnnotation {
    }

    @SourceAnnotation
    static int a = 1;

    @Retention(RetentionPolicy.CLASS)   // optional, Default
    @interface ClassAnnotation {
    }

    @ClassAnnotation
    static int b = 2;

    @Retention(RetentionPolicy.RUNTIME)
    @interface RuntimeAnnotation {
    }

    @RuntimeAnnotation
    static int c = 3;
}
```

Listing 9.18: Annotationen mit allen drei Arten von Retention.

Nach dem Übersetzen zeigt der Disassembler das Innenleben des Bytecodes. Der Schalter `-v` ist nötig, um die interessanten Einzelheiten sichtbar zu machen:

```
$ javap -v AnnotationLevels
...
  #18 = Utf8               LAnnotationLevels$ClassAnnotation;
...
  #21 = Utf8               LAnnotationLevels$RuntimeAnnotation;
...
  static int a;
    flags: ACC_STATIC

  static int b;
    flags: ACC_STATIC
    RuntimeInvisibleAnnotations:
      0: #18()

  static int c;
    flags: ACC_STATIC
    RuntimeVisibleAnnotations:
      0: #21()
...
```

Man erkennt, dass bei `a` keine Informationen über Annotationen aufgezeichnet sind. Bei `b` ist unter `RuntimeInvisibleAnnotations` ein Verweis auf den Typ mit

dem Index 18 vermerkt, der gemäß der vorhergehenden Liste `AnnotationLevels$ClassAnnotation` ist. Entsprechend findet sich bei `c` unter `RuntimeVisibleAnnotations` der Verweis auf Eintrag 21 in der Konstantenliste, wo `AnnotationLevels$RuntimeAnnotation` steht.

Ähnliche Informationen liefert der Java-Compiler selbst mit dem Schalter `-Xprint`. Er arbeitet hier alleine mit dem Bytecode und braucht keinen Quelltext:

```
$ javac -Xprint AnnotationLevels
public class AnnotationLevels {

  @java.lang.annotation.Retention(java.lang.annotation.RetentionPolicy.RUNTIME)
  static @interface RuntimeAnnotation {
  }

  @java.lang.annotation.Retention(java.lang.annotation.RetentionPolicy.CLASS)
  static @interface ClassAnnotation {
  }

  @java.lang.annotation.Retention(java.lang.annotation.RetentionPolicy.SOURCE)
  static @interface SourceAnnotation {
  }
  static int a;
  @AnnotationLevels.ClassAnnotation
  static int b;
  @AnnotationLevels.RuntimeAnnotation
  static int c;

  public AnnotationLevels();
}
```

Man erkennt auch hier, dass `b` und `c` annotiert sind, `a` aber nicht.

Mit dem gleichen Schalter kann der Java-Compiler auch Quelltext analysieren und gibt dann auch Javadoc-Kommentare mit aus. ◀

9.3.5 Meta-Annotation `Inherited` und `Documented`

Zwei weitere Meta-Annotationen stehen für eigene Annotationen zur Verfügung.

`@Inherited`

Annotationen und Vererbung

Eine so markierte Annotation vererbt sich automatisch auf abgeleitete Klassen. Diese Meta-Annotation ist nur bei Annotationen mit Ziel `Target.TYPE` sinnvoll. Bei anderen Annotationen ist sie syntaktisch zulässig, wird aber ignoriert.

`@Documented`

In der Voreinstellung erfährt der Anwender eines Programmelements nichts über die Annotationen der entsprechenden Definition im Quelltext.

Für `@Override` und `@SuppressWarnings` ist das sinnvoll, weil sie die Übersetzung des Quelltexts betreffen und für den Anwender des fertigen Codes keine Rolle mehr spielen.

Anders liegt die Sache bei `@Deprecated` und `@SafeVarargs`. Der Anwender einer Methode oder Klasse, die abgekündigt ist, sollte das erfahren. Entsprechend sollte der Nutzer einer `@SafeVarargs`-Methode darüber informiert werden, dass er sie gefahrlos verwenden kann.

Annotationen in generierter Dokumentation

Die Meta-Annotation `@Documented` weist aus, ob die Anwesenheit einer Annotation von öffentlichem Interesse ist oder nicht. Die konkrete Umsetzung der Meta-Annotation bleibt allerdings Sache einzelner Werkzeuge und ist nicht zwingend vorgeschrieben. Javadoc berücksichtigt beispielsweise `@Documented` und weist so markierte Annotationen in der generierten Dokumentation aus.

Anwendungsbeispiele für `@Inherited` und `@Documented` folgen auf Seite 589.

9.3.6 Marker-Annotationen

Annotationen ohne Attribute

Annotationen ohne Attribute mit Target `ElementType.TYPE` und Retention `RetentionPolicy.SOURCE` werden als **Marker-Annotationen** bezeichnet. Sie spielen eine ähnliche Rolle wie Marker-Interfaces, wie zum Beispiel `Serializable` (Seite 129), deren bloße *Anwesenheit* gewisse Eigenschaften einer Definition dokumentiert.

Marker-Annotationen vs. Marker-Interfaces

Den Zweck der Dokumentation erfüllen Annotationen sogar besser als Marker-Interfaces. Interfaces implizieren Aussagen über Typen und Operationen, die im Falle von Marker-Interfaces allerdings nicht existieren. Daraus ergibt sich die Frage, wie ein Objekt eines Typs *ohne Operationen* eigentlich verwendet werden kann. Marker-Interfaces eignen sich damit zwar technisch zur Dokumentation bestimmter Eigenschaften, konzeptionell dagegen eher nicht.

Marker-Annotationen haben darüber hinaus gegenüber Marker-Interfaces den Vorteil, dass sie nicht zwangsläufig auf abgeleitete Klassen übertragen werden. Das Verhalten lässt sich mit der Meta-Annotation `@Inherited` nach Bedarf steuern.[11]

[11] Noch mehr Freiheit gibt ein `boolean`-Attribut, das eine Annotation wahlweise aktiviert oder stilllegt. Formal ist die Annotation dann allerdings keine Marker-Annotation mehr.

9.4 Auswertung zur Laufzeit

Zugriff auf Annotationen via Reflection

Annotationen mit der Retention `RUNTIME` überträgt der Compiler in den Bytecode und die JVM in das laufende Programm. Um diese Annotationen auszuwerten eignet sich Reflection (Kapitel 8).

Ausgehend von einem beliebigen Objekt `x` liefert die Methode `getClass` das zugeordnete Typobjekt, das den Typ von `x` repräsentiert:

```
Class<?> type = x.getClass();
```

Das Typobjekt öffnet den Zugang zu den Bestandteilen der Typdefinition. Das Package `java.lang.reflect` definiert Klassen für die verschiedenen Bestandteile:

```
Package pkg = type.getPackage();
Constructor<?>[] ctors = type.getDeclaredConstructors();
Method[] methods = type.getDeclaredMethods();
Field[] fields = type.getDeclaredFields();
```

Die folgende Methode `dissectObject` zerlegt den Typ eines beliebigen Objekts `x` in seine Elemente und ruft mit jedem Element eine Methode `printAnnotations` auf, die weiter unten definiert wird:

```
static void dissectObject(Object x) {
    Class<?> type = x.getClass();
    System.out.printf("%s: %s%n", type.getSimpleName(), x);
    printAnnotations(type.getPackage());
    printAnnotations(type);
    for(Constructor<?> ctor: type.getDeclaredConstructors())
        printAnnotations(ctor);
    for(Method method: type.getDeclaredMethods())
        printAnnotations(method);
    for(Field field: type.getDeclaredFields())
        printAnnotations(field);
}
```

Listing 9.19: Typ eines Objekts zerlegen und Annotationen aller Elemente ausgeben.

Interface für annotierbare Codeelemente

Alle Bestandteile des Typs, das heißt die Objekte der Typen `Class`, `Package`, `Constructor`, `Method` und `Field`, können annotiert sein. Entsprechend implementieren alle das Interface `AnnotatedElement` mit den folgenden Auskunftsmethoden:

`Annotation[] getAnnotations()`
: Liefert die öffentlichen Annotationen dieser Definition.

`Annotation[] getDeclaredAnnotations()`
: Wie `getAnnotations`, liefert aber *alle* Annotationen gemäß Quelltext.

`<T extends Annotation> T getAnnotation(Class<T> type)`
: Liefert die Annotation vom Typ `type` oder `null`, wenn es keine solche gibt.

`boolean isAnnotationPresent(Class<? extends Annotation> type)`
: Gibt Auskunft, ob die Definition eine Annotation vom Typ `type` hat oder nicht. Diese Methode überprüft nur die *Anwesenheit* einer bestimmten Annotation. Einzelheiten der betreffenden Annotation liefert sie nicht.

Beispiel: Auflisten von Annotationen

Die oben verwendete Methode `printAnnotations` wird hier definiert. Sie erwartet ein Argument des Typs `AnnotatedElement`, holt sich dessen Annotationen und gibt sie aus. Das Argument `null` ignoriert sie:

```
static void printAnnotations(AnnotatedElement element) {
    if(element != null)
        for(Annotation annotation: element.getDeclaredAnnotations())
            System.out.printf("\t%s: %s%n", element, annotation);
}
```

Listing 9.20: Annotationen eines annotierten Elementes ausgeben.

Die folgende Anwendung ruft die beiden Methoden `dissectObject` und mittelbar auch `printAnnotations` auf, um die Annotationen einiger Objekte sichtbar zu machen:

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

public class ReflectOnObject {
    public static void main(String... args) {
        dissectObject(args);
        dissectObject(args[0]);
        dissectObject(new Thread());
        dissectObject(Math.PI);
        dissectObject(new AnnotationLevels());
        dissectObject(new SafeVarargsAnnotation());
    }

    // static void dissectObject(Object x) ...

    // static void printAnnotations(AnnotatedElement ae) ...
}
```

Listing 9.21: Annotationen einiger Objekte per Reflection suchen und ausgeben.

Der Aufruf fördert eine Anzahl `@Deprecated`-Annotationen zutage, dazu die `RUNTIME`-Annotationen von `AnnotationLevels` (Listing 9.18) und `SafeVarargsAnnotation` (Listing 9.13). Die Ausgabe ist um Package-Angaben gekürzt:

```
$ java ReflectOnObject Hello
String[]: [LString;@6f9f88
String: Hello
        public String(byte[],int,int,int): @Deprecated()
        public String(byte[],int): @Deprecated()
        public void String.getBytes(int,int,byte[],int): @Deprecated()
Thread: Thread[Thread-0,5,main]
        public native int Thread.countStackFrames(): @Deprecated()
        public void Thread.destroy(): @Deprecated()
        public final void Thread.resume(): @Deprecated()
        public final void Thread.stop(): @Deprecated()
        public final synchronized void Thread.stop(Throwable): @Deprecated()
        public final void Thread.suspend(): @Deprecated()
Double: 3.141592653589793
AnnotationLevels: AnnotationLevels@16c02df
        static int AnnotationLevels.c: @AnnotationLevels$RuntimeAnnotation()
SafeVarargsAnnotation: SafeVarargsAnnotation@127a1d8
        static void SafeVarargsAnnotation.printAll(Object[]): @SafeVarargs()
```

9.4.1 Anwendung: Reflektive `toString`-Methode

In Kapitel 8.3.4 wurde eine statische Methode entwickelt, die ein beliebiges Objekt mit Reflection durchleuchtet und versucht, aus den Variablenwerten eine brauchbare String-Darstellung zu konstruieren. Die Implementierung in `ReflectiveToString` geht allerdings mangels genauerer Informationen etwas unspezifisch vor und zieht *alle* Variablen im Objekt gleichermaßen in Betracht.

Verbesserung einer früheren Lösung

Definition des Annotationtyps `IntoString`

Annotationen erlauben eine genauere Kontrolle über die String-Darstellung von Objekten. Hier wird eine Annotation `IntoString` entwickelt, die einzelne Objekt- und Klassenvariablen zur Aufnahme in eine String-Darstellung ausweist. `IntoString` zielt auf Objekt- und Klassenvariablen (`@Target = FIELD`) und wird zur Laufzeit gebraucht (`@Retention = RUNTIME`):

Festlegen von Ziel und Lebensdauer

```
import java.lang.annotation.*;
import java.util.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface IntoString {
```

```
    Class<?> toStringSupplier() default Objects.class;
}
```

Listing 9.22: Markiert eine Definition zur Aufnahme in die String-Darstellung.

Das Annotation-Attribut `toStringSupplier` ermöglicht die Steuerung der String-Darstellung einzelner Variablen. Der Attributwert muss eine Klasse sein, die die folgende Methode definiert:[12]

```
static String toString(Object x)
```

Das Attribut ist optional. Wenn es fehlt, fällt die Annotation auf die Klasse `java.util.Objects` zurück. `Objects` definiert die geforderte statische `toString`-Methode, die mit `null` zurechtkommt und ansonsten lediglich die universelle Methode `Object.toString` aufruft.

Anwendung der Beispiel-Annotation

Die folgende Beispielklasse `BankAccount` nutzt die Annotation `IntoString`. Die Variablen `name` und `number` sind damit annotiert, `pin` dagegen nicht.

```
class BankAccount {
    @IntoString
    private final String name;

    @IntoString(toStringSupplier = StarredNumber.class)
    private final int number;

    private final int pin;

    public BankAccount(String name, int number) {
        this.name = name;
        this.number = number;
        pin = (int)(Math.random()*1000);
    }
}
```

Listing 9.23: Klasse mit privaten Objektvariablen.

Damit wird verhindert, dass schützenswerte Geheimzahlen beispielsweise in Protokolldateien auftauchen. In schwächerem Maße gilt das auch für Kontonummern. Deshalb übernimmt die Klasse `StarredNumber` Ausgabe von Kontonummern. Ihre statische `toString`-Methode ersetzt bei Zahlen einen Teil der Ziffern durch Sterne und reicht andere Objekte an `Objects` weiter:[13]

[12] Ein Test, ob die angegebene Klasse tatsächlich die geforderte Methode definiert, unterbleibt an dieser Stelle.

[13] Der `instanceof`-Test in dieser Methode ist nicht gerade elegant. Statische Methoden nehmen am dynamischen Binden allerdings nicht teil und haben daher kaum eine Wahl.

```
import java.util.*;

public class StarredNumber {
    public static String toString(Object x) {
        return x instanceof Integer?
               "*****" + ((int)x%1000):
               Objects.toString(x);
    }
}
```

Listing 9.24: `toString`-Methode, die bei Zahlen einige Ziffern verschleiert.

Auswertung durch Reflection

Die neue reflektive `toString`-Methode ähnelt weitgehend der früheren Fassung `ReflectiveToString`. Geändert ist der Rumpf der `for`-Schleife, die die Variablen durchläuft. Die Änderungen sind im Code kommentiert.

Annotationen zur Laufzeit suchen

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

public class ReflectiveAnnotatedToString {
    public static String toString(Object x) throws ReflectiveOperationException {
        return toString(x, x.getClass());
    }

    private static String toString(Object x, Class<?> type) throws ReflectiveOperationExc
        if(x == null)
            return "null";
        if(type == null || type == Object.class)
            return "";
        if(type.isArray())
            return Arrays.toString((Object[])x);
        if(type.isEnum() || type == String.class)
            return x.toString();

        String string = toString(x, type.getSuperclass());
        for(Field field: type.getDeclaredFields()) {
            // Annotation IntoString suchen
            IntoString annot = field.getAnnotation(IntoString.class);
            if(annot == null)
                continue;   // keine Annotation? Variable auslassen
            // Wert des Annotation-Attributs toStringSupplier
            Class<?> annotType = annot.toStringSupplier();
            // 'toString'-Methode lokalisieren (NoSuchMethodException falls nicht gefunde
            Method method = annotType.getMethod("toString", Object.class);
            field.setAccessible(true);
            // toString mit Wert der Variablen aufrufen
            Object value = method.invoke(null, field.get(x));
            string += ", " + field.getName() + "=" + value;
        }
        if(string.startsWith(", "))
            string = string.substring(2);
        return type.getSimpleName() + "(" + string + ")";
```

```
        }
    }
```

Listing 9.25: Von Annotation gesteuerte, reflektive `toString`-Methode.

Die folgende `main`-Methode erzeugt ein paar Bankkonten und gibt sie aus:

```
public static void main(String[] args) throws ReflectiveOperationException {
    System.out.println(toString(new BankAccount("Duck, Dagobert", 12345678)));
    System.out.println(toString(new BankAccount("Mouse, Minnie", 22334455)));
}
```

Listing 9.26: Test der Ausgabesteuerung mit Annotationen.

Sie liefert die gewünschte Ausgabe mit gekürzten Kontonummern und fehlenden Geheimzahlen:

```
$ java ReflectiveAnnotatedToString Hello
BankAccount(name=Duck, Dagobert, number=*****678)
BankAccount(name=Mouse, Minnie, number=*****455)
```

Erweiterung auf Klassen-Annotation

Erweiterung der Anwendbarkeit auf Klassen

In der bisher entwickelten Fassung muss jede Variable, die in der String-Darstellung erscheinen soll, einzeln mit `@IntoString` annotiert werden. Das ist unter Umständen etwas mühsam und lässt sich vereinfachen: Die Erweiterung von `IntoString` auf ganze Klassen wirkt pauschal auf alle enthaltenen Variablendefinitionen.

Zum Ziel `ElementType.FIELD` kommt in der Definition von `IntoString` (Listing 9.22) noch `ElementType.TYPE` hinzu. Beim Attributwert von `@Target` muss jetzt zur Array-Schreibweise mit geschweiften Klammern gewechselt werden.

```
@Target({ElementType.FIELD, ElementType.TYPE})
```

Auswertung von Annotation-Attributen

Damit ist `@IntoString` syntaktisch auch vor Typdefinitionen zulässig. Die reflektive `toString`-Methode in `ReflectiveAnnotatedToString` (Listing 9.25) weiß davon allerdings nichts und reagiert auch nicht auf die Annotation des Typs. Die folgende Erweiterung ermittelt zuerst die `IntoString`-Annotation `classAnnot` der ganzen Klasse und fällt auf diese Annotation zurück, wenn eine Variable keine eigene Annotation `annot` trägt:

```
private static String toString(Object x, Class<?> type) throws ReflectiveOperationExcepti
    if(x == null)
        return "null";
    if(type == null || type == Object.class)
        return "";
    if(type.isArray())
        return Arrays.toString((Object[])x);
    if(type.isEnum() || type == String.class)
        return x.toString();
    // Klassen-Annotation suchen
    IntoString classAnnot = type.getAnnotation(IntoString.class);
    String string = toString(x, type.getSuperclass());
    for(Field field: type.getDeclaredFields()) {
        IntoString annot = field.getAnnotation(IntoString.class);
        // Keine Variablen-Annotation? Klassen-Annotation verwenden
        if(annot == null)
            annot = classAnnot;
        if(annot == null)
            continue;
        Class<?> annotType = annot.toStringSupplier();
        Method method = annotType.getMethod("toString", Object.class);
        field.setAccessible(true);
        Object value = method.invoke(null, field.get(x));
        string += ", " + field.getName() + "=" + value;
    }
    if(string.startsWith(", "))
        string = string.substring(2);
    return type.getSimpleName() + "(" + string + ")";
}
```

Listing 9.27: Rückfall auf die Klassen-Annotation bei fehlender Variablen-Annotation.

Als Anwendungsbeispiel dient eine von `BankAccount` abgeleitete Klasse `SavingsBankAccount`, die komplett mit `@IntoString` annotiert ist:

```
@IntoString
public class SavingsBankAccount extends BankAccount {
    private final double rate;

    public SavingsBankAccount(String name, int number, double rate) {
        super(name, number);
        this.rate = rate;
    }
}
```

Listing 9.28: Anwendung einer Annotation auf eine ganze Klasse und damit auf alle Objektvariablen.

Der Aufruf

```
toString(new SavingsBankAccount("Duck, Dagobert",
                                12345678,
                                0.04))
```

ergibt den folgenden String, in dem die Objektvariable `rate` erfasst ist (Abdruck zur besseren Lesbarkeit umgebrochen):

```
SavingsBankAccount(
    BankAccount(name=Duck, Dagobert, number=*****678),
    rate=0.04)
```

Einsatz von `@Inherited`

Vererbbare Annotation

Versieht man die Definition von `IntoString` (Listing 9.22) noch zusätzlich mit der Meta-Annotation `@Inherited`, dann vererbt sich `IntoString` auf abgeleitete Objekte.

Das betrifft eine von `SavingsBankAccount` weiter abgeleitete Klasse `TemporarySavingsBankAccount`:

```
public class TemporarySavingsBankAccount extends SavingsBankAccount {
    private final String expires;

    public TemporarySavingsBankAccount(String name, int number, double rate, String expir
        super(name, number, rate);
        this.expires = expires;
    }
}
```

Listing 9.29: Vererbung einer `Inherited`-Annotation.

Der Ausdruck

```
toString(new TemporarySavingsBankAccount("Mouse, Minnie",
                                         22334455,
                                         0.04,
                                         "2015-01-01"))
```

liefert den String (Abdruck zur besseren Lesbarkeit umgebrochen):

```
TemporarySavingsBankAccount(
    SavingsBankAccount(
        BankAccount(name=Mouse, Minnie, number=*****455),
        rate=0.04),
    expires=2015-01-01)
```

Er enthält die Objektvariable `expires` der Klasse `TemporarySavingsBankAccount`, in der selbst keine Annotation `@IntoString` vorkommt. `TemporarySavingsBankAccount` erbt die Klassen-Annotation aber von der Basisklasse `SavingsBankAccount`, weil sie mit `@Inherited` definiert ist. Entfernt man `@IntoString` aus `SavingsBankAccount`, dann ändert sich das Ergebnis zu:

```
TemporarySavingsBankAccount(
    SavingsBankAccount(
        BankAccount(name=Mouse, Minnie, number=*****455)))
```

Weder `rate` noch `expire` erscheinen in diesem String, weil `toString` keine `@IntoString`-Annotation für diese Variablen findet.

9.5 Prozessoren

Annotationen mit der Retention `RUNTIME` lassen sich zur Laufzeit mit Reflection auswerten, wie der vorhergehende Abschnitt zeigt. Dagegen sind Annotationen mit der Retention `SOURCE` nur dem Compiler zugänglich.

9.5.1 Arbeitsweise

Compiler ruft Annotationprozessoren auf

Der Java-Compiler verarbeitet `SOURCE`-Annotationen nicht selbst, sondern ruft externe **Annotationprozessoren** auf, die aber auf den Ablauf der Übersetzung Einfluss nehmen können. Annotationprozessoren sind Java-Klassen. Sie können beispielsweise

- den aktuellen Quelltext untersuchen und ihr Veto gegen die weitere Übersetzung einlegen,
- zusätzlichen Quelltext erzeugen, der dann mitübersetzt wird,
- neuen Bytecode erzeugen, der sofort nutzbar ist,
- bereits erzeugten Bytecode analysieren.

Kein Entfernen oder Ändern von Code

Annotationprozessoren können *nicht* den gerade übersetzten Quelltext ändern oder austauschen. Diese Einschränkung spiegelt die Rolle von Annotationen als *Zusatzinformation* wider, die die grundsätzliche Arbeitsweise des annotierten Konstrukts nicht beeinträchtigt.

Als Beispiel soll die folgende Annotation dienen:

```
public @interface HelloWorld {
}
```

Listing 9.30: Minimale Annotation zur Verarbeitung durch einen Prozessor.

Die folgende Klasse ist mit `HelloWorld` annotiert:

```
@HelloWorld public class AnnotatedClass {
}
```

Listing 9.31: Anwendung einer Annotation zur Verarbeitung durch einen Prozessor.

Einfacher Prozessor

ABC für Annotationprozessoren

Annotationprozessoren sind von der ABC `AbstractProcessor` abgeleitet. Die Ablaufumgebung von Prozessoren ist der Compiler, der sie zwischen der Syntaxanalyse und der Codegenerierung aufruft.

Die Klassen-Annotation `@SupportedAnnotationTypes` legt fest, für welche Annotationen sich ein Prozessor interessiert. Die wichtigste Methode eines Prozessors ist

```
boolean  process(Set<? extends TypeElement> annots,
                 RoundEnvironment env)
```

Dabei ist `annots` die Menge der Annotationen gemäß `SupportedAnnotationTypes` und `env` eine Abstraktion des Zustandes des Compilers. Das Ergebnis von `process` signalisiert, ob der Prozessor die Annotationen verarbeitet hat (`true`) oder ob sie an einen anderen Prozessor weitergegeben werden sollen (`false`).

Minimaler Annotationprozessor

Die folgende Prozessor-Klasse behandelt `HelloWorld`-Annotationen.

```
import java.util.*;
import javax.annotation.processing.*;
import javax.lang.model.*;
import javax.lang.model.element.*;

@SupportedAnnotationTypes("HelloWorld")
public class HelloWorldProcessor extends AbstractProcessor {
    public boolean process(Set <? extends TypeElement > elements, RoundEnvironment round)

        System.out.println("Hello, World!");
        return true;
    }
}
```

Listing 9.32: Minimaler Annotation-Prozessor.

Compileraufruf mit Annotationprozessor

Vor dem ersten Einsatz muss der neue Prozessor übersetzt werden:

```
$ javac HelloWorldProcessor.java
```

Der Schalter `-processor` des Compilers bindet den Prozessor jetzt zur Verarbeitung von Annotationen ein. Die doppelte Ausgabe wird unten erklärt.[14]

Compileraufruf mit Annotationprozessor

```
$ javac -processor HelloWorldProcessor AnnotatedClass.java
Hello, World!
Hello, World!
$
```

Die Ausgabe bleibt aus, wenn man den gleichen Quelltext ohne Prozessor übersetzt. Der Java-Compiler ignoriert Annotationen, für die sich kein Prozessor zuständig erklärt:

Annotationen ohne Prozessor werden ignoriert

```
$ javac AnnotatedClass.java
$
```

`process` wird möglicherweise mehrmals nacheinander aufgerufen. Ein Prozessor kann neuen Quelltext generieren, der wiederum Annotationen enthält. Dann startet eine neue „Runde“ der Verarbeitung und so fort, bis schließlich alle Annotationen verarbeitet sind. Am Ende folgt noch eine letzte Runde, in der alle Prozessoren Gelegenheit zu Aufräumarbeiten erhalten. Diese Abschlussrunde lässt sich mit dem Getter `RoundEnvironment.processingOver` erkennen, der dann das Ergebnis `true` liefert.

Mehrere Verarbeitungsrunden

Die doppelte Ausgabe im vorletzten Beispiel lässt sich vermeiden, wenn die letzte Runde ausgelassen wird:

```
public boolean process(Set <? extends TypeElement > elements, RoundEnvironment round) {
    if(round.processingOver())  // Letzte Runde? keine Aktion
        return true;

    System.out.println("Hello, World!");
    return true;
}
```

Listing 9.33: Prozessor, der in der letzten Runde sofort abbricht.

▶ Sie erhalten möglicherweise bei Aufruf des Prozessors eine Warnung der Art:

```
warning: No SupportedSourceVersion annotation found on HelloWorldProcessor, returning REL
warning: Supported source version 'RELEASE_6' from annotation processor 'HelloWorldProces
```

[14] Die Warnung „No SupportedSourceVersion“ erklärt der nachfolgende Exkurs.

Diese Warnung verursacht die Basisklasse `AbstractProcessor`, die in der Laufzeitbibliothek von Java 7 nur Java 6 als höchste unterstützte Sprachversion meldet.

Dem lässt sich auf zwei Arten abhelfen: Zum einen können Sie die Definition des `HelloWorldProcessor` mit der Meta-Annotation

```
@SupportedSourceVersion(SourceVersion.RELEASE_7)
```

versehen. Zum anderen kann die Basisklassenmethode im `HelloWorldProcessor`

```
SourceVersion  getSupportedSourceVersion()
```

so redefiniert werden, dass sie als Ergebnis die Konstante `javax.lang.model.SourceVersion.RELEASE_7` liefert. ◀

9.5.2 Diagnoseausgabe des Compilers

Ausgaben als Compilermeldungen

Die Ausgabe auf `System.out` folgt nicht dem Schema der anderen Ausgaben des Compilers. Der Aufruf von

```
processingEnv.getMessager(). printMessage(Kind.NOTE, "text");
```

fügt eine Nachricht in die regulären Diagnoseausgaben des Compilers ein. Dabei ist `processingEnv` eine `protected`-Variable[15] der Basisklasse `AbstractProcessor`. Der erste Parameter von `printMessage` vom Typ `javax.tools.Diagnostic.Kind` steuert die Wichtigkeit der Nachricht. Abgesehen von `Kind.NOTE` stehen noch `WARNING` und `ERROR` zur Verfügung.

Ersetzt man im `HelloWorldProcessor` die Ausgabe auf die Standardausgabe durch `printMessage`, dann erhält man die folgende Ausgabe:[16]

```
$ javac -processor HelloWorldProcessor AnnotatedClass.java
Note: Hello, World!
```

9.5.3 Anwendung: Generierte Bean-Klassen

Getter und Setter für Objektvariablen

Viele Klassen stellen geschützte Objektvariablen über öffentliche Getter und Setter

[15] Diese Konstruktion ist nicht gerade elegant.

[16] Äußerlich wirkt der Unterschied banal. Der Compiler kann aber in einem ganz anderen Kontext laufen als auf der Kommandozeile. Diagnoseausgaben erscheinen dann nicht auf dem Bildschirm, sondern finden auch dort den richtigen Adressaten.

gemäß Beans-Konventionen (siehe Seite 147) zur Verfügung. Die entsprechenden Methoden sind recht einfach gebaut und lassen sich oft mechanisch erzeugen.

Eine Annotation `Property` soll diese Arbeit automatisieren. Das Ziel von `Property` sind Objektvariablen. Das drückt die Meta-Annotation `@Target(FIELD)` aus.[17] Weiter sollen die Getter und Setter im Zuge der Übersetzung entstehen und damit anderen Klassen sofort zur Verfügung stehen. Als Retention reicht daher `@Retention(SOURCE)` aus.

```
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.SOURCE)
public @interface Property {
}
```

Listing 9.34: Annotation für Objektvariablen, zu denen Getter und Setter erzeugt werden sollen.

Ein Anwendungsbeispiel zeigt die folgende Definition. Sie enthält nur annotierte Objektvariablen. Abgesehen von den Annotationen ist die Klasse ziemlich nutzlos.[18]

```
public class PlayerParts {
    @Property String name;

    @Property int score;
}
```

Listing 9.35: Mit `Property` annotierte Objektvariablen zur Expansion in einem Prozessor.

Zusätzliche Klasse

Ein Annotationprozessor kann diese bestehende Klassendefinition nicht mit zusätzlichen Methoden ausstatten. Er kann aber eine *neue* Klasse mit den weiteren Methoden generieren. Der Name der neuen Klasse ergibt sich aus dem Namen der annotierten Klasse durch Austausch der Namensendung ...`Parts` durch ...`Bean`. Aus `PlayerParts.java` erzeugt der Prozessor beispielsweise die Klasse `PlayerBean` (Listing 2.18).

Zur Vereinfachung gelten Einschränkungen:

[17] `@Target(FIELD)` trennt nicht zwischen Objekt- und Klassenvariablen.

[18] Allerdings verfügen Objekte der Klasse durchaus über Methoden: Der Compiler generiert ungefragt einen Default-Konstruktor mit dem gleichen Zugriffsschutz wie die Klasse selbst, in diesem Beispiel `public`. Weiter erbt die Klasse die `Object`-Methoden `equals`, `toString`, `wait` und so weiter.

- Der Prozessor berücksichtigt nur annotierte Variablen und wirft alle anderen Elemente der annotierten Klasse weg.
- Modifier der verarbeiteten Variablen gehen verloren. Die Bean-Klasse bildet sie alle als private Objektvariablen ab.

Der `Property`-Prozessor arbeitet in zwei Schritten: Zuerst sammelt er die annotierten Variablen in der Menge `varNodes` ein, dann generiert er mit der Methode `generateBean` daraus den Quelltext der Bean-Klasse.

```
001 import java.io.*;
002 import java.util.*;
003 import java.util.regex.*;
004 import javax.annotation.processing.*;
005 import javax.lang.model.element.*;
006 import javax.naming.*;
007 import javax.tools.*;
008
009 @SupportedAnnotationTypes("Property")
010
011 public class PropertyProcessor extends AbstractProcessor {
012     @Override public boolean process(Set <? extends TypeElement > annots, RoundEnviro
013         Element classNode = null;
014         Messager messager = processingEnv.getMessager();
015         Set<VariableElement> varNodes = new HashSet<>();
016         for(TypeElement annotation: annots)
017             for(Element node: env.getElementsAnnotatedWith(annotation)) {
018                 varNodes.add((VariableElement)node);
019                 classNode = node.getEnclosingElement();
020             }
021         if(!varNodes.isEmpty())
022             try {
023                 messager.printMessage(Diagnostic.Kind.NOTE, varNodes.toString());
024                 generateBean(classNode, varNodes);
025             }
026             catch(IOException | InvalidNameException ex) {
027                 messager.printMessage(Diagnostic.Kind.ERROR, ex.getMessage());
028             }
029
030         return true;
031     }
032
033     // Definition von generateBean ...
034 }
```

Listing 9.36: Prozessor für `Property`-Annotationen.

Arbeitsschritte des Annotationprozessors

Im Einzelnen geht der Prozessor folgendermaßen vor:

- 013: `classNode` speichert die compilerinterne Repräsentation der ganzen Klassendefinition als Objekt vom Typ `Element`.

- 015: Die Menge `varNodes` nimmt die annotierten Variablendefinitionen auf, die als Objekte vom Typ `VariableElement` repräsentiert sind.
- 016: Die vom Compiler im Parameter `annotations` übergebene Menge aller Annotationen enthält hier nur nur ein einziges Element, nämlich die Annotation `Property`, weil sich dieser Prozessor mit der Annotation `SupportedAnnotationTypes` für keine anderen Annotationen angemeldet hat.
- 017: Die Methode `getElementsAnnotatedWith` liefert die annotierten Codeelemente, in diesem Fall ausschließlich Variablendefinitionen.
- 018: Jede einzelne Definition wird vorerst in `varNodes` gespeichert. Der Typecast auf den Typ `VariableElement` ist sicher, weil gemäß `Target`-Meta-Annotation in `Property` (Listing 9.34) nichts anderes als Variablen annotiert sein kann.
- 019: Bei dieser Gelegenheit gerät der Knoten der gesamten Klassendefinition bequem in Reichweite. In der Schleife wird zwar immer wieder derselbe Wert an `classNode` zugewiesen, weil alle Variablendefinitionen in derselben Klasse stecken, aber hier lohnt sich keine Performance-Optimierung.
- 021: Wenn annotierte Variablendefinitionen gefunden wurden, generiert der Prozessor eine Bean-Klasse. Wenn es überhaupt keine Annotationen gab, wird nichts generiert.
- 024: Der Aufruf von `generateBean` erhält die Klassendefinition und alle Variablendefinitionen.
- 026: Falls die Methode beim Generieren der Bean-Klasse scheitert, bricht der Compiler ab.

Die folgende Methode `generateBean` ist verantwortlich für das Erzeugen der Bean-Klasse:

```
001 private static final Pattern partsClassnamePattern =
002     Pattern.compile("((.*)\\.)?(.+)Parts");
003
004 private void generateBean(Element classNode, Set<VariableElement> varNodes)
005 throws IOException, InvalidNameException {
006     Matcher matcher = partsClassnamePattern.matcher(classNode.toString());
007     if(!matcher.matches())
008         throw new InvalidNameException("class name must end in: Parts");
009     String pkg = matcher.group(2);
010     String className = matcher.group(3) + "Bean";
011     String fullName = pkg == null? className: pkg + '.' + className;
012     try(Writer writer = processingEnv.getFiler().createSourceFile(fullName).openWrite
013         PrintWriter printWriter = new PrintWriter(writer)) {
014         if(pkg != null)
015             printWriter.printf("package %s;%n", pkg);
016         printWriter.printf("public class %s {%n", className);
017         for(VariableElement varNode: varNodes) {
018             String varType = varNode.asType().toString();
019             String varName = varNode.getSimpleName().toString();
020             String capName = Character.toUpperCase(varName.charAt(0))
021                              + varName.substring(1);
```

```
022             printWriter.printf("\tprivate %s %s;%n", varType, varName);
023             printWriter.printf("\tpublic %s %s%s() {%n",
024                                varType,
025                                varType.equals("boolean")? "input": "get",
026                                capName);
027             printWriter.printf("\t\treturn %s;%n", varName);
028             printWriter.printf("\t}%n");
029             printWriter.printf("\tpublic void set%s(%s %s) {%n",
030                                capName,
031                                varType,
032                                varName);
033             printWriter.printf("\t\tthis.%s = %s;%n", varName, varName);
034             printWriter.printf("\t}%n");
035         }
036         printWriter.printf("}%n");
037     }
038 }
```

Listing 9.37: Generieren des Quelltextes der Bean-Klasse.

- 002–007: Aus dem qualifizierten Namen der annotierten Klasse ergibt sich der Package-Pfad (`pkg`, falls nicht leer) und der Name der Bean-Klasse (`className`).
- 003: Der Prozessor erwartet, dass Name der annotierten Klasse mit `Parts` endet, sonst bricht er den Compiler ab.[19]
- 008: Der Getter `getFiler` der Variablen `processingEnv` liefert die Filesystem-Umgebung, in der der Compiler arbeitet. `createSourceFile` öffnet in dieser Umgebung einen Writer für eine Quelltextdatei, die der Compiler anschließend aufgreift und mitübersetzt.
- 010–025: Der Schleifenrumpf produziert den Quelltext der generierten Bean-Klasse. Dazu ist nur etwas Stringverarbeitung und formatierte Ausgabe nötig.[20]

Der Prozessor wird zuerst übersetzt:

```
$ javac PropertyProcessor.java
```

Die vom Prozessor generierten Quelltexte übersetzt der Compiler sofort, sodass sie in Anwendungen zur Verfügung stehen:[21]

[19] Das ist nicht mehr als eine Schutzmaßnahme, um dem Leser klarzumachen, dass die `Parts`-Klasse nur eine Vorstufe der „wirklichen“ Klasse darstellt.

[20] Mit weniger Code könnte man schlechter lesbaren Quelltext generieren, den der Compiler aber ebenso akzeptiert.

[21] Es ist nicht nötig die Quelltextdateien in einer bestimmten Reihenfolge anzugeben. Der Compiler erkennt selbst, dass erst Annotationen zu verarbeiten sind, bevor er die übrigen Quelltextdateien angeht.

```
$ javac -processor PropertyProcessor PlayerBeanMain.java PlayerParts.java
Note: [score, name]
$ ls PlayerBean.*
PlayerBean.class
PlayerBean.java
$ java PlayerBeanMain max.xml
```

Die Beispielanwendung `PlayerBeanMain` (Seite 149) serialisiert oder deserialisiert eine `PlayerBean` in XML-Darstellung und benutzt dazu die Getter und Setter. Sie arbeitet reibungslos mit der generierten Klasse `PlayerBean` zusammen.

Trennung von Quellen und Artefakten

In den vorhergehenden Aufrufbeispielen geraten generierte und „echte“ Quelltextdateien, die sich nicht mechanisch herleiten lassen, durcheinander. Mit dem Compilerschalter `-s` *path* können die Primärdaten von automatisch erzeugten Daten getrennt werden. Er legt *path* als Ziel-Directory fest, in das der Compiler prozessorgenerierte Artefakte ablädt. Im folgenden Beispiel landen generierte Quelltexte im neu erstellten Directory `generated`:

```
$ mkdir generated
$ javac -s generated -processor PropertyProcessor PlayerBeanMain.java PlayerParts.java
Note: [score, name]
$ ls generated
PlayerBean.class
PlayerBean.java
$ java -cp .:generated PlayerBeanMain max.xml
$ java -cp .:generated PlayerBeanMain max.xml read
Max
5
```

9.6 Packages

Packages mit Klassen zur Annotation-Verarbeitung

Bei der Verarbeitung von Annotationen kommen viele Klassen aus der Bibliothek ins Spiel. Diese Klassen sind über verschiedene Packages verteilt, die hier noch einmal zusammengestellt sind:

`java.lang.annotation`
: Typen für jede Art von Annotationen, Definition von Meta-Annotationen und Aufzählungstypen ihrer Werte.

`java.lang.reflect`
: Klassen und Methoden zur Untersuchung von Objekten auf `RUNTIME`-Annotationen.

`javax.lang.model` und Subpackages
: Typen zur Abbildung von Programmbausteinen. Notwendig zur Untersuchung von `SOURCE`-annotierten Quelltextelementen in Annotationprozessoren, die im Compiler ablaufen.

`com.sun.source` und Subpackages
: Typen zur Repräsentation der vollständigen compilerinternen Programmdarstellung.[22] Diese Klassen sind zur genauen Analyse des Quelltexts nötig, weil die Typen in `javax.lang.model` (noch) nicht jeden Aspekt eines Programms abbilden. Dieses Package ist herstellerspezifisch und seine Verwendung nicht portabel.

Zusammenfassung

- **Annotationen** spielen eine ähnliche Rolle wie **Modifier**. Sie können mit **Attributen** ergänzt werden.
- In Java 7 können **Definitionen** (Typen, Methoden, Variablen) **annotiert** werden.
- `@Deprecated`, `@Override`, `@SafeVarargs` und `@SuppressWarnings` sind **vordefinierte Annotationen**.
- Der **Compiler** reagiert auf Annotationen möglicherweise mit **Warnungen,** die mit **Schaltern ein- und ausgeblendet** werden können.
- **Neue Annotationen** und ihre Attribute werden ähnlich wie Interfaces **definiert** und in Bytecode übersetzt.
- **Meta-Annotationen** (`@Target`, `@Retention`) steuern, **wo** Annotationen platziert werden dürfen und **wie lange** sie erhalten bleiben.
- Annotationen lassen sich **zur Laufzeit mit Reflection** aufspüren und auswerten.
- **Annotationprozessoren** laufen *im Compiler* ab und können **Annotationen im Quelltext** auswerten. Prozessoren werden beim Aufruf des Compilers explizit angefordert.
- Prozessoren können während der Übersetzung **zusätzlichen Quelltext generieren**, den der Compiler sofort übersetzt, aber keinen bestehenden Quelltext verändern oder löschen.

[22] Diese Darstellung wird im Compilerbau als AST (*Abstract Syntax Tree*) bezeichnet. Sie enthält den geparsten Quelltext, aus dem anschließend der Bytecode entsteht.

Aufgaben

Aufgabe 1: Mock-Klassen

Die Lösung einer Übungsaufgabe zum Thema Reflection (Seite 561) generiert aus Interfaces Mock-Klassen zum Testen. Ein anderer Weg führt über eine neue Annotation `@Mock`, mit der Interfaces versehen werden.

Definieren Sie den Annotation-Typ `Mock` und schreiben Sie einen Annotationprozessor `MockProcessor`, der diese Annotation beim Übersetzen eines Interface erkennt und sofort eine Mock-Klasse mit dem gleichen Aufbau wie in der früheren Aufgabe erzeugt.

Beispielsweise ist das folgende Interface `Student` mit `@Mock` annotiert:

```
@Mock public interface Student {
    String getLastname();

    String getFirstname();

    int getId();
}
```

Listing 9.38: Interface mit einer `Mock`-Annotation, aus der eine Mock-Klasse generiert wird.

Beim Übersetzen mit dem `MockProcessor`:

```
$javac -processor MockProcessor Student.java
```

entsteht aus dem Interface die Klasse `MockStudent.java`, die sofort verwendet werden kann:

```
public class MockStudent implements Student {
    public int getId() {
        System.out.printf("getId()%n");
        return 0;
    }
    public java.lang.String getLastname() {
        System.out.printf("getLastname()%n");
        return null;
    }
    public java.lang.String getFirstname() {
        System.out.printf("getFirstname()%n");
```

```
            return null;
        }
    }
```

Listing 9.39: Aus dem annotierten Interface vom Annotationprozessor generierte Mock-Klasse.

Diese Lösung hat gegenüber der rein reflektiven Lösung den Vorteil, dass Generierung und Übersetzung der Mock-Klasse vom Compiler im Hintergrund abgewickelt werden. Die Lösung hat den Nachteil, dass der Quelltext des Interface zur Verfügung stehen und angepasst werden muss.

Nutzen Sie hier Reflection (Kapitel 8). Die nächste Aufgabe zeigt eine andere Art von Reflection.

Aufgabe 2: Analyse der Codestruktur

Die Annotation `Immutable` soll eine Klasse als unveränderlich ausweisen. Diese Eigenschaft ist im Allgemeinen schwer nachzuweisen. In dieser Aufgabe geht es daher nur um eine vereinfachte Sicht von „Unveränderlichkeit“. Eine Klasse ist sicherlich unveränderlich, wenn

1. alle Objektvariablen `final` und
2. die Typen der Objektvariablen primitiv oder `String` sind.

Entwickeln Sie einen Prozessor, der diese Eigenschaften bei `Immutable`-annotierten Klassen überprüft und einen Fehler meldet, wenn sie nicht zutreffen.

Der Prozessor muss Variablendefinitionen untersuchen, die nicht einzeln mit Annotationen versehen sind und daher nicht direkt zur Verfügung stehen. Stattdessen dient ein Visitor-Objekt vom Typ `ElementScanner7`[23] zum Besuchen der untergeordneten Programmelemente. Das folgende Programm zeigt eine minimale Implementierung eines entsprechenden Prozessors. Im Visitor gibt es Methoden, die beim Erreichen verschiedener Programmbausteine aufgerufen werden:[24]

[23] Die „7“ im Namen zeigt an, dass in dieser ABC Methoden für Quelltextstrukturen von Java 7 definiert sind. Die entsprechende Klasse `ElementScanner6` eignet sich für Quelltexte von Java-Version 6. In künftigen Java-Versionen wird es wahrscheinlich weiter abgeleitete Klassen mit neuen Methoden geben, die kommende Erweiterungen erfassen.

[24] Die Visitor-Methoden `visitPackage` und `visitTypeParameter` wurden der Kürze wegen weggelassen. Die Methode `visitUnknown` ist ein Fallback für künftige Java-Versionen, in denen weitere Elemente besucht werden. Die Implementierung von `visitUnknown` in `ElementScanner7` wirft eine `UnknownElementException`.

`visitVariable`
: Für Parameter, Objekt- und Klassenvariablen,

`visitExecutable`
: für Methoden, einschließlich Konstruktoren und

`visitType`
: für Interface- und Klassendefinitionen.

```
import java.util.*;
import javax.annotation.processing.*;
import javax.lang.model.*;
import javax.lang.model.element.*;
import javax.lang.model.util.*;

@SupportedAnnotationTypes("AnAnnotation")
@AnAnnotation
public class BasicVisitorProcessor extends AbstractProcessor {
    @Override public boolean process(Set <? extends TypeElement > elements, RoundEnvironm
        ElementScanner7<Object, Object> scanner7 = new ElementScanner7<Object, Object>()
            @Override public Object visitVariable(VariableElement variable, final Object
                System.out.println("variable: " + variable);
                return super.visitVariable(variable, x);
            }

            @Override public Object visitExecutable(ExecutableElement executable, Object
                System.out.println("executable: " + executable);
                return super.visitExecutable(executable, x);
            }

            @Override public Object visitType(TypeElement type, Object x) {
                System.out.println("type: " + type);
                return super.visitType(type, x);
            }
        };
        for(TypeElement typeElement: elements)
            for(Element element: env.getElementsAnnotatedWith(typeElement))
                scanner7.scan(element);
        return true;
    }
}
```

Listing 9.40: Rahmen eines Annotation-Prozessors mit Visitor.

Dieses Programm ist mit der trivialen Annotation `AnAnnotation` versehen, die es selbst verarbeitet. Man kann es auf sich selbst anwenden und erhält beim Übersetzen des eigenen Quelltexts die folgende Ausgabe:

```
$ javac -processor BasicVisitorProcessor BasicVisitorProcessor.java
type: BasicVisitorProcessor
executable: BasicVisitorProcessor()
```

```
executable: process(Set<? extends TypeElement>,RoundEnvironment)
variable: elements
variable: env
```

Ein `ElementScanner7` kann den inneren Aufbau von Methodenrümpfen nicht „sehen“.

Bauen Sie den `BasicVisitorProcessor` zum `ImmutableProcessor` aus. Mit dem `ImmutableProcessor` soll der Compiler beispielsweise die annotierten Klassen `Rational` (Listing 9.15) und `BankAccount` (Listing 9.23) fehlerfrei übersetzen, weil alle Objektvariablen `final` und primitiv oder `String` sind. Stellen Sie sicher, dass der Annotationprozessor die Übersetzung einer `Immutable`-annotierten Klasse ohne `final` oder mit anderen Typen verweigert.

Aufgabe 3: Annotation-Attribute

Erweitern Sie die `Property`-Annotation (Abschnitt 9.5.3) um ein `boolean`-Attribut `mutable` mit dem Defaultwert `false`.

Der Prozessor erkennt dieses Attribut. Für `mutable = true` generiert er Code für eine veränderliche Objektvariable, wie bisher. Beim Wert `false` generiert er Code für eine unveränderliche Objektvariable. Das bedeutet im Einzelnen:

- Die Variablendefinition erhält den Modifier `final`.
- Es gibt keinen Setter.
- Der Prozessor generiert einen Konstruktor.
- Der Konstruktor verfügt über eine Parameterliste mit einem Parameter für jede unveränderliche Objektvariable.
- Die unveränderlichen Objektvariablen erhalten im Konstruktor ihre Werte aus korrespondierenden Parametern.

Das folgende Beispiel zeigt eine Anwendung des `mutable`-Attributs:

```
public class PlayerParts {
    @Property String name;
    @Property(mutable=true) int score;
}
```

Der Prozessor soll daraus eine Klasse entsprechend zu `Player` (Listing 2.21) generieren.

Erweitern Sie die `Property`-Annotation außerdem um ein zweites `boolean`-Attribut namens `Indexed`, das ebenfalls den Defaultwert `false` hat. Der erweiterte Prozessor setzt eine so annotierte Variable in eine indexierte Property um. Aus der Definition (mit Plural-s im Namen)

```
public class PlayersParts {
    @Property(indexed=true) PlayerBean players;
}
```

entsteht eine Klassendefinition nach dem Schema von `PlayersBean` (Listing 2.19). Berücksichtigen Sie im Annotationprozessor auch, dass die beiden Annotationen `Mutable` und `Indexed` kombiniert werden können.

Anhang

A Voraussetzungen

Die Themen dieses Buches setzen einige Java-Kenntnisse voraus, die hier in aller Kürze zusammengestellt sind.

Primitive numerische Typen, Variablen, Arithmetik
: Java kennt acht **primitive Typen**, die bis auf einen numerisch sind. Zwischen den Typen gibt es **implizite Typumwandlungen**, mit **Typecasts** können explizite Typumwandlungen erzwungen werden. Die Arithmetik ist **polymorph** und zeigt an den **Wertebereichsgrenzen** unterschiedliches Verhalten.

Wahrheitswerte und Kontrollstrukturen
: **Relationale** und **logische Operatoren** arbeiten mit `boolean`**-Werten**, die die meisten **Kontrollstrukturen** steuern (Alternativen und Schleifen). Lokale Variablen sind in statischen **Gültigkeitsbereichen** definiert und existieren zur Laufzeit in möglicherweise vielen Inkarnationen.

Klassen
: Klassendefinitionen schaffen neue **Typen** und dienen der Modularisierung. Sie umfassen in erster Linie **Objektvariablen** und **Methoden**. Objekte werden mit **Konstruktoren** geschaffen, ein Sonderfall von Methoden. Methoden bestehen aus einem Kopf und einem Rumpf. Beim Aufruf können **Argumente** in **Parametern** übergeben und ein **Ergebniswert** kann zurückgegeben werden.
  Als **Referenztypen** haben Klassen grundsätzlich andere Eigenschaften als primitive Typen. Insbesondere kommt es bei unabhängigen Referenzen auf dasselbe Objekt zu **Aliasing**, das heißt zu Seiteneffekten bei Änderungen.
  Datenkapselung verbirgt Objektvariablen mit `private` und macht nur Methoden sichtbar. **Unveränderliche Klassen** haben Vorteile, kosten aber unter Umständen Ressourcen. **Klassenvariablen** und **statische Methoden** gelten global und unabhängig von Objekten. Vergleich und Kopieren von Objekten führen zur Frage von **Identität** und **Gleichheit** sowie von flachen und tiefen Operationen.

Textzeichen und Strings

Der primitive **Typ** `char` repräsentiert ein Textzeichen. Zeichen sind in **Zeichensätzen** gesammelt, die Zeichen auf Codes abbilden. Java verwendet **Unicode**. Nützliche Methoden zum Umgang mit Zeichen sind als statische Methoden der **Wrapperklasse** `Character` definiert. Es gibt zu jedem primitiven Typ eine korrespondierende Wrapperklasse.

Die **Klasse** `String` repräsentiert Zeichenfolgen. `String` ist unveränderlich und genießt einen Sonderstatus. Objekte können mit **String-Literalen** definiert werden, zur Konkatenation ist der **Additionsoperator** überladen. Die Methode `toString` produziert eine String-Darstellungen beliebiger Objekte und wird oft implizit aufgerufen. Aus Effizienzgründen kann der Einsatz der regulären Klasse `StringBuilder` sinnvoll sein, die „veränderliche“ Strings implementiert.

Arrays

Arrays sind **Containertypen**, die eine lineare Anordnung von Elementen speichern. Zu jedem Java-Typ gibt es einen korrespondierenden **Arraytyp.** Arraytypen sind Referenztypen, aber keine Klassen. Einzelne Elemente werden mit einem Null-basierten **Index** angesprochen. Mit `for`- oder *foreach*-Schleifen können alle Elemente eines Arrays nacheinander durchlaufen werden. Bei Arrays tritt der Unterschied zwischen **flachen und tiefen Operationen** deutlich zutage, insbesondere beim Kopieren und Vergleichen.

Mehrdimensionale Arrays sind als Arrays von Arrays implementiert. Das lässt nicht rechteckige Arrays zu.

Methoden können mit einem **Vararg-Parameter** definiert werden, der eine beliebige Anzahl Argumente erlaubt. Die Argumente werden als Array übergeben.

Vererbung

Java trennt zwischen Klassen und **Interfaces.** Interfaces definieren nur **Methodenköpfe** und enthalten keinen Code. Sie müssen von Klassen **implementiert** werden, die Methoden mit den geforderten Köpfen enthalten. Interfaces sind **Typen**, zu denen alle implementierenden Klassen **kompatibel** sind. An Interface-Variablen können Objekte jeder implementierenden Klasse zugewiesen werden. Methodenaufrufe werden in Java **dynamisch gebunden**. Anwendungen können (sollen!) mit Interface-Typen arbeiten und werden damit unabhängig von konkreten Klassen. Das ist der entscheidende Beitrag zur **Modularisierung.**

Von einer Klasse können weitere Klassen **abgeleitet** werden. Das kann mehrstufig geschehen und führt zu **Vererbungshierarchien.** Abgeleitete Klassen **erben** die Methoden der **Basisklasse** und können sie bei Bedarf **redefinieren** oder zusätzliche Methoden definieren, die die Basisklasse nicht kennt. Abgeleitete Klassen sind zu Basisklassen **kompatibel** und

Methodenaufrufe werden **dynamisch gebunden**.
Java lässt nur **einfache Vererbung** zu. Das heißt, dass eine Klasse nur *eine* Basisklasse haben kann. Darüber hinaus kann sie aber beliebig viele Interfaces implementieren.
Abstrakte Basisklassen (*Abstract Base Class*, ABC) definieren einige konkrete Methoden und einige **abstrakte Methoden**. Sie *müssen* abgeleitet werden, wobei letztlich alle abstrakten Methoden definiert werden müssen. ABCs können gemeinsame Funktionalität verschiedener konkreter Klassen zusammenfassen und helfen, kopierten Code zu vermeiden. ABCs teilen einige Eigenschaften mit Interfaces, sind aber dennoch vollwertige Klassen.
Alle Klassen erben implizit von der **Klasse** `Object`, der einzigen Klasse ohne Basisklasse. `Object`-Methoden werden an alle Klassen vererbt und stehen in allen Objekten zur Verfügung. Einige `Object`-Methoden *sollten* redefiniert werden, darunter `toString` und insbesondere `equals` und `hashCode`, für deren Aufbau es bewährte Idiome gibt.

Packages Packages dienen der inneren **Organisation** größerer Java-Programme. Sie entsprechen etwa Directories in einem Filesystem. Jede Klasse ist Teil eines Packages und legt das mit einer **Package-Klausel** fest. Um Klassen anderer Packages zu verwenden, werden entweder **qualifizierte Namen** oder **Import-Klauseln** benutzt. Das **anonyme Defaultpackage** enthält alle Klassen ohne Package-Klausel. **Statische Import-Klauseln** binden die statischen Elemente einer anderen Klasse ein.
Java kennt vier Ebenen des Zugriffsschutzes. Die Voreinstellung (ohne Modifier) erlaubt Zugriff innerhalb eines Packages.
Klassen können in **Jar-Dateien** gepackt und damit leichter transportiert werden. Jar-Dateien sind technisch Zip-Dateien mit einigen Metadaten. Zur Arbeit mit Jar-Dateien dient das **Kommandozeilenwerkzeug** `jar`. Die JVM kann Bytecode gleichermaßen im Filesystem und in Jar-Dateien lokalisieren.

Dokumentation
Javadoc-Kommentare verankern Dokumentation im Quelltext. Sie sind vor Klassen, Variablen und Methoden erlaubt. Ihr Aufbau ist halb formal, sie enthalten neben Fließtext **Tags** mit Informationen zu bestimmten Aspekten der dokumentierten Definition. Der **Javadoc-Compiler** generiert aus Javadoc-Kommentaren Dokumentation als HTML-Seiten oder in anderen Formaten.

*Zusicherungen (*Assertion*) und Exceptions*
Zusicherungen sind `assert`-Anweisungen mit einer Bedingung, die unter allen Umständen gilt. Zusicherungen müssen beim Start eines Programms **aktiviert** werden. Sollte eine Zusicherung nicht gelten, dann

bricht das Programm ab. Zusichern lassen sich Bedingungen, deren Einhaltung das Programm durch eigene **Logik garantiert** und die nicht vom Anwender oder von anderen äußeren Einflüssen abhängen. Dazu gehören insbesondere **Klasseninvarianten** und **Postconditions** am Methodenende.

Exceptions sind Objekte, die ein Programm bei Problemen mit `throw` wirft. Eine Exception unterbricht den normalen Ablauf und muss **aufgefangen** oder **weitergegeben** werden. Zum Auffangen wird der kritische Code in einen `try`-Block gesetzt, dieser wird von `catch`-Blöcken gefolgt. Jedes `catch` gilt für bestimmte Exceptiontypen und ignoriert alle anderen. Zum Weitergeben werden die potenziellen Exceptions in der **Exceptionsignatur** im Methodenkopf aufgezählt. **Exceptions** sichern unter anderem **Preconditions** ab.

Nur von `Throwable` abgeleitete Objekte können geworfen werden. `Error` und abgeleitete Klassen sind **JVM-internen Ursachen** vorbehalten. `RuntimeException` und abgeleitete Klassen signalisieren **fehlerhaften Code.** Behebbare Probleme repräsentieren Klassen, die von `Exception` abgeleitet sind.

`Errors` und `RuntimeExceptions` sind **Unchecked-Exceptions**. Sie sollten nicht gefangen werden und können in der Exceptionsignatur weggelassen werden. Alle anderen sind **Checked-Exceptions** und müssen aufgefangen oder in der Exceptionsignatur genannt werden.

Ein `finally`-Block nach `try/catch` wird unter allen Umständen ausgeführt.

Collections

Das **Collection-Framework** definiert nützliche Containerklassen, darunter Container mit Einzelelementen (Interface `Collection`) und Elementpaaren (Interface `Map`). Als Elemente sind nur Referenztypen erlaubt, aber **Autoboxing** und **Auto-Unboxing** ebnen den Weg für primitive Werte über Wrapperklassen. Wrapperklassen führen zu heiklen Identitätsfragen.

Zu den `Collection`-Klassen zählen **Listen** (`List`) und **Mengen** (`Set`) mit verschiedenen konkreten Implementierungen (`ArrayList` und `LinkedList`, `HashSet` und `TreeSet`). Sie können mit **Iteratoren** durchlaufen werden. Das **Interface** `Iterable` macht sie *foreach*-Schleifen zugänglich. Die häufigsten Maps sind `HashMap` und `TreeMap`.

Im Zusammenhang mit Containerklassen sind die Methoden `equals` und `hashCode` wichtig. Eine „schlechte“ Implementierung kann ineffizienten Code bedeuten. `TreeSet` und `TreeMap` sortieren ihre Elemente der Größe nach. Eine Ordnung definiert das Interface `Comparable`. Alternative Ordnungen legen Klassen fest, die das Interface `Comparator` implementieren.

Die Utility-Klasse `Collections` bietet statische Methoden mit gängigen

Algorithmen für Container, wie zum Beispiel sortieren, suchen, mischen.

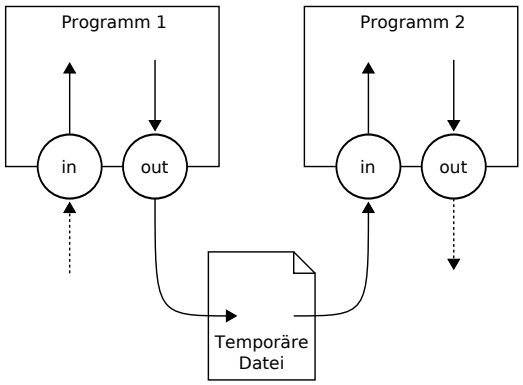
Generics Klassen, die sich bei gleichem Code nur in Typen unterscheiden (beispielsweise Container im Collection-Framework), können als **generische Klasse** mit **Typvariablen** definiert werden. Der Anwender bildet daraus mit einem konkreten **Typargument** einen **generischen Typ**, der wie jeder andere Typ verwendet werden kann. Mit **Typebounds** kann eine generische Klasse die späteren Typargumente einschränken und dafür selbst die Funktionalität des Typebounds nutzen.
Generische Typen derselben generischen Klasse sind inkompatibel, auch wenn die Typargumente kompatibel sind. **Wildcards** heben diese Inkompatibilität kontrolliert auf, schränken dafür aber die möglichen Operationen ein, um Typsicherheit zu gewährleisten.
Der Java-Compiler übersetzt generischen Code durch **Type-Erasure** in normalen Code. Die JVM weiß nichts von Generics. Diese **fehlende Laufzeit-Typinformation** zieht Einschränkungen nach sich. Zum Beispiel kann ein Typparameter nicht als Basisklasse, als Elementtyp eines Arrays oder für einen Konstruktoraufruf verwendet werden. Manche Beschränkungen können mit Ersatzkonstruktionen umgangen werden.
Neben ganzen Klassen können auch einzelne **Methoden generisch** sein. Die **Type-Inference** des Compilers macht die Angabe von Typargumenten beim Aufruf generischer Methoden oft unnötig.

Anhang

B I/O-Pipelines

Einige der im Kapitel 1 entwickelten Programme holen sich Eingaben von der Standardeingabe und schreiben Ergebnisse auf die Standardausgabe. Beispiele sind `ConsoleEcho`, `Detab` und `LineCharCounter`. Oft sollen zwei solche Programme nacheinander angewendet werden, wobei die Ausgabe des ersten Programms als Eingabe des zweiten dient. Beispielsweise könnte man zuerst die Tabulatoren eines Textes mit `Detab` expandieren und das Ergebnis dann mit `LineCharCounter` ausmessen. Das Zwischenergebnis des ersten Programms muss dabei als temporäre Datei irgendwo „geparkt“ werden:

Verkettung von Programmen mit Standardein- und -ausgabe



Auf Unix eignet sich dafür das Directory `/tmp`, das als „Halde“ für alle Arten von temporären Dateien dient:[1]

Temporäre Dateien für Zwischenergebnisse

```
$ java Detab < Detab.java > /tmp/notabs
$ java LineCharCounter < /tmp/notabs
...
```

[1] Die System-Property `java.io.tmpdir` zeigt, welches Directory auf Ihrem System für diesen Zweck vorgesehen ist. Der Aufruf `java -XshowSettings:properties` listet alle System-Propertys auf, darunter auch `java.io.tmpdir`.

Wo genau dabei die temporäre Datei `notabs` abgespeichert wird und wie sie heißt, spielt dabei keine Rolle. Die Datei wird ja nur als kurzzeitiges „Zwischenlager“ gebraucht. Am Besten wird sie nachher wieder gelöscht, um keinen nutzlosen „Datenmüll“ zu hinterlassen und um anderen Programmen nicht in die Quere zu kommen.

Pipelines ersetzen temporäre Dateien

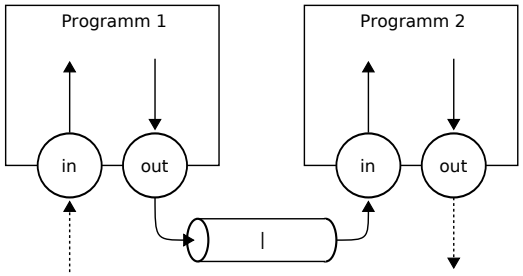
Mit **Pipelines** kann dieser Ablauf elegant vereinfacht werden. Die beiden Einzelaufrufe

```
programm1 > tempdatei
programm2 < tempdatei
```

lassen sich durch den Aufruf

```
programm1 | programm2
```

ersetzen. Diese Konstruktion wird als Pipeline (im Sinne von „Rohrleitung“) bezeichnet, das Symbol | heißt „Pipe-Symbol“.



Eine Pipe funktioniert tatsächlich wie eine Rohrleitung: Die Standardausgabe des ersten Programms wird über ein unsichtbares Rohr direkt mit der Standardeingabe des zweiten Programms verbunden. Alle Ausgaben des Programms vor dem Pipe-Symbol verschwinden im „Rohr“ und kommen am anderen Ende wieder zum Vorschein, wo sie als Eingabe des Programms nach dem Pipe-Symbol dienen. Die durch die Pipe fließenden Daten werden dabei in keiner Datei abgespeichert.[2] Das vorhergehende Beispiel lässt sich damit kürzen zu

```
$ java Detab < Detab.java | java LineCharCounter
...
```

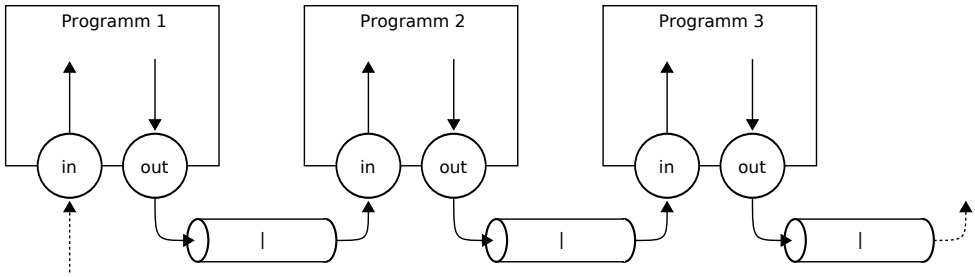
[2] Ein Betriebssystem *kann* die Pipeline über das Dateisystem abwickeln. Der Benutzer muss sich darum aber nicht mehr kümmern, die Verwaltung ist alleine Sache des Betriebssystems.

Das Ergebnis ist das gleiche wie vorher, allerdings ohne eine Zwischendatei wie `/tmp/notabs`. Es muss weder ein Directory für die Zwischendatei gefunden noch ein Name erfunden werden.

Pipelines mit mehreren Programmen

Diese Konstruktion lässt sich fortsetzen. Die Standardausgabe des zweiten Programms kann über ein weiteres Pipe-Symbol mit einem dritten Programm gekoppelt werden, dieses mit einem vierten und so weiter. Das Ergebnis ist eine Pipeline aus vielen Programmen.

```
programm1 | programm2 | programm3 | ...
```



Im folgenden Beispiel wird die Ausgabe von `LineCharCounter` selbst vermessen. Das ist zwar nicht besonders sinnvoll, verdeutlicht aber das Prinzip:

```
$ java Detab < Detab.java | java LineCharCounter
1463 Zeichen, 46 Zeile(n)
$ java Detab < Detab.java | java LineCharCounter | java LineCharCounter
26 Zeichen, 1 Zeile(n)
$ java Detab < Detab.java | java LineCharCounter | java LineCharCounter | java LineCharCo
24 Zeichen, 1 Zeile(n)
```

Filter

Filter lesen Standardeingabe, schreiben Standardausgabe

Auf diesem einfachen Prinzip beruhen viele Kommandozeilenwerkzeuge von Unix. Viele lesen die Standardeingabe und schreiben auf die Standardausgabe. Sie sind zur Konstruktion von Pipelines gedacht. Derartige Programme werden auch als **Filter** bezeichnet, weil sie „durchfließende“ Daten wie eine Flüssigkeit nach bestimmten Kriterien „filtern“, anreichern oder auf andere Art umformen.

Weil praktisch alle Programmiersprachen Standardein- und -ausgabe unterstützen, können in einer Pipeline Programme kombiniert werden, die in unterschiedlichen Sprachen implementiert sind. Ebenso lassen sich selbst entwickelte Filter mit den vom System bereitgestellten Filtern zusammenfügen. Im folgenden Beispiel wird

die Ausgabe des Unix-Dienstprogramms `ls`, das den Inhalt des Arbeitsdirectory auflistet[3], mit `LineCharCounter` vermessen:

```
$ ls | java LineCharCounter
125 Zeichen, 6 Zeile(n)
```

`ls` gibt jeden Filenamen[4] auf einer eigenen Zeile aus. Im Arbeitsdirectory befinden sich in diesem Beispiel also 6 Files.

Parallelverarbeitung und Multicore-CPUs

Eine wichtige Eigenschaft von Pipelines ist nicht sofort erkennbar: Die Filter arbeiten zeitlich überlappend. Das bedeutet, dass das Betriebssystem einen nachfolgenden Filter schon startet, sobald der vorhergehende die ersten Daten liefert. Ein Filter wartet insbesondere *nicht*, bis der vorhergehende Filter seine Arbeit beendet und alle Daten geliefert hat. Das Gleiche gilt für die weiteren Filter in der Pipeline. Wenn genügend Daten durchgesetzt werden, arbeiten also möglicherweise alle Filter der Pipeline zur gleichen Zeit an unterschiedlichen Punkten im Datenstrom.[5] Diese Eigenschaft ist besonders mit Blick auf die inzwischen weitverbreiteten Multicore-CPUs interessant, die sich durch Pipelines mit einfachsten Mitteln gut auslasten lassen.

Nachweis der Parallelverarbeitung

Die parallele Arbeitsweise weist eine künstlich gebremste Version von `ConsoleEcho` nach. Das Programm `SlowConsoleEcho` legt nach jedem kopierten Zeichen eine kurze Pause von zehn Millisekunden ein:

```
import java.io.*;
import static java.lang.System.*;

public class SlowConsoleEcho {
    public static void main(String... args) throws IOException, InterruptedException {
        for(int code = in.read(); code >= 0; code = in.read()) {
            out.write(code);
            out.flush();
            Thread.sleep(10);
        }
    }
}
```

Listing B.1: Verzögertes Kopieren der Standardeingabe auf die Standardausgabe.

3 Auf Windows liefert das Kommando `dir /b` eine ähnliche Ausgabe.

4 Diese Aussage ist vereinfacht. In einem Unix-Filesystem finden sich nicht nur Files und Directories, sondern noch andere, zum Teil exotische Bewohner. In diesem Beispiel werden alle Elemente des Arbeitsdirectory gezählt, welcher Art auch immer sie sind.

5 Tatsächlich parallel arbeiten können höchstens so viele Filter, wie das System an Prozessoren bereitstellt. Andernfalls teilt das Betriebssystem die verfügbaren Prozessoren zeitweise zu, sodass äußerlich der Eindruck gleichzeitiger Verarbeitung entsteht. Dieses Thema wird in Kapitel 6 (Seite 361) ausführlich diskutiert.

`SlowConsoleEcho` hat einen maximalen Durchsatz von 100 Zeichen pro Sekunde. Aus der Länge der Eingabedatei kann man die voraussichtliche Laufzeit ungefähr abschätzen.[6]

```
$ java SlowConsoleEcho < SlowConsoleEcho.java > /dev/null
```

Die Laufzeit liegt im Bereich von einigen Sekunden und kann zum Beispiel mit einer Uhr grob nachgemessen werden.[7] Die Zieldatei ist dabei belanglos.[8] Wegen der parallelen Arbeitsweise steigt die Rechenzeit nur unwesentlich an, wenn man zwei oder mehr der gleichen Filter in einer Pipeline hintereinanderschaltet:

```
$ java SlowConsoleEcho < SlowConsoleEcho.java | java SlowConsoleEcho | java SlowConsoleEc
```

Würden die einzelnen Filter der Pipeline nicht parallel, sondern sequenziell ablaufen, dann ergäbe sich ein Vielfaches der vorher gemessenen Einzellaufzeit.

Noch deutlicher wird der parallele Ablauf, wenn die durchfließenden Daten zusätzlich auf der Standard-Fehlerausgabe protokolliert werden:

```
import java.io.*;
import static java.lang.System.*;

public class SlowStderrEcho {
    public static void main(String... args) throws IOException, InterruptedException {
        for(int code = in.read(); code >= 0; code = in.read()) {
            out.write(code);
            out.flush();
            err.write(code);
            err.flush();
            Thread.sleep(10);
        }
    }
}
```

Listing B.2: Gleichzeitiges und verzögertes Kopieren der Standardeingabe auf die Standardausgabe und auf die Fehlerausgabe.

In einer Pipeline mischen sich die Ausgaben aller Filter auf `System.err`. Der folgende Aufruf zeigt den Anfang der Ausgabe:

---

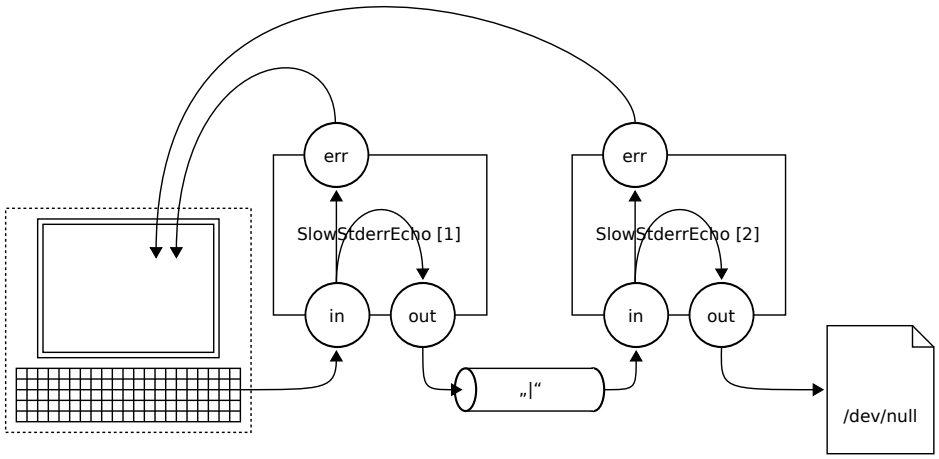
6 Die hochgerechneten Laufzeiten stimmen nicht genau, weil zum Beispiel der Start der JVM selbst Zeit beansprucht. Dazu kommen noch die übrigen Aktivitäten des Systems, das mit anderen Aufgaben mehr oder weniger beschäftigt sein kann.

7 Eine genauere Messung der Laufzeit erlaubt das Unix-Systemkommando `time`. Darum geht es hier aber nicht.

8 `/dev/null` ist keine echte Datei, sondern ein Unix-Pseudofile. Was immer in diese „Datei“ geschrieben wird, wird einfach verworfen. Die Datei nimmt keinen Platz ein, unabhängig davon, wie viel Daten darauf geschrieben werden. Auf Windows kann die Zieldatei `nul` verwendet werden.

```
$ java SlowStderrEcho < SlowStderrEcho.java | java SlowStderrEcho > /dev/null
iimmppoorrtt  jjaavvaa..iioo..**;;

iimmppoorrtt  ssttaattiicc  jajvaav.al.alnagn.gS.ySsytsetme.m*.;*

p
upbulibcl iclca scsl SalsoswS tSdleorwrSEtcdheor r{E
...
```

Mit der Zeit kommen die Ausgaben aus dem Tritt, weil die Filter unabhängig laufen und nicht genau im Takt bleiben. Im vorhergehenden Beispiel geschieht das in der dritten Ausgabezeile ab „jajvaav“. Die folgende Skizze zeigt den Fluss der Daten:



Anhang

C Beispiel Decorator-Pattern

Pizzas mit Boden und Auflagen

Das Decorator-Pattern eignet sich als Lösungsansatz für ganz unterschiedliche Problemstellungen. In diesem Anhang soll das Muster an einem konkreten Beispiel veranschaulicht werden, das nichts mit Ein- und Ausgabe zu tun hat.

Ein moderner Pizzadienst möchte seinen Betrieb mit Software unterstützen. Als Teil dieses Projekts sollen die verschiedenen Pizzas im Angebot durch Objekte modelliert werden. An diesem Beispiel wird das Decorator-Pattern plastisch umgesetzt.

Interface für alle Bausteine

Gemeinsame Eigenschaften aller Pizzas sind die Größe, der Preis sowie die Angaben, ob sie vegetarisch und ob sie scharf sind. Entsprechende Methoden werden in einem Interface festgelegt:

```
public interface Pizza {
    int getPrice();

    boolean isVegetarian();

    boolean isHot();
}
```

Listing C.1: Interface Pizza.

Grundlage und Dekoratoren

Als konkrete Bausteine dienen Pizzaböden, die „einfachsten“ möglichen Pizzas. Die gemeinsamen Eigenschaften von Pizzaböden (Preis, scharf oder nicht) fasst die abstrakte Basisklasse[1] `Base` zusammen. Alle Pizzaböden sind vegetarisch, Preis und Schärfe werden jeweils im Konstruktor festgelegt:

```
public abstract class Base implements Pizza {
    private final int price;
```

[1] Die Pizzaböden in diesem einfachen Beispiel sind sich so ähnlich, dass sogar ein Aufzählungstyp ausreichen würde.

```
    private final boolean hot;

    public Base(int price, boolean hot) {
        this.price = price;
        this.hot = hot;
    }

    public int getPrice() {
        return price;
    }

    public boolean isHot() {
        return hot;
    }

    public boolean isVegetarian() {
        return true;
    }
}
```

Listing C.2: Aufzählungstyp als konkrete Komponentenklasse: Pizzaböden.

Von `Base` sind drei Varianten von Pizzaböden abgeleitet:

```
public class Crunchy extends Base {
    public Crunchy() {
        super(300, false);
    }
}
```

Listing C.3: Knuspriger Pizzaboden.

```
public class Puffy extends Base {
    public Puffy() {
        super(400, false);
    }
}
```

Listing C.4: Weicher, dicker Pizzaboden.

```
public class Sicilian extends Base {
    public Sicilian() {
        super(350, true);
    }
}
```

Listing C.5: Scharfer Pizzaboden.

Abstrakter Dekorator

Die verschiedenen Auflagen werden als Dekoratoren implementiert. Die gemeinsame Eigenschaft aller Auflagen, die Verwaltung der darunterliegenden Pizza, wird in eine gemeinsame abstrakte Basisklasse `Topping` herausgezogen. `Topping` implementiert das Interface `Pizza` und enthält eine Objektvariable vom Typ `Pizza`, die die darunterliegende restliche Pizza ohne diese Auflage repräsentiert. Die Klasse ist abstrakt, obwohl sie keine abstrakten Methoden enthält. Damit können keine Instanzen erzeugt werden. Alle Methoden des Interface werden an die Objektvariable delegiert.

```
public abstract class Topping implements Pizza {
    private final Pizza below;

    public Topping(Pizza below) {
        this.below = below;
    }

    public boolean isVegetarian() {
        return below.isVegetarian();
    }

    public boolean isHot() {
        return below.isHot();
    }

    public int getPrice() {
        return below.getPrice();
    }
}
```

Listing C.6: Abstrakte Dekoratorklasse: Pizzaauflage.

Konkrete Dekoratoren

Von der abstrakten Basisklasse `Topping` können verschiedene konkrete Dekoratoren abgeleitet werden, die die ererbten Methoden nach Bedarf redefinieren

Jede Auflage Käse erhöht den Preis der gesamten Pizza um 1 Euro. Sie ändert nichts an den Eigenschaften „vegetarisch“ und „scharf“. Die aus der Basisklasse ererbten Methoden `isVegetarian` und `isHot` rufen die entsprechenden Eigenschaften der restlichen Pizza, ohne diesen Käse, ab und geben das Ergebnis an den Aufrufer zurück. Beim Preis wird der Preis dieser Käseauflage (1 Euro) zum Preis der übrigen Pizza addiert und die Summe als Preis der ganzen Pizza (dieser Käse mit allem anderen) zurückgegeben.

```
public class Cheese extends Topping {
    public Cheese(Pizza below) {
        super(below);
    }

    public int getPrice() {
```

```
        return 100 + super.getPrice();
    }
}
```

Listing C.7: Konkrete Dekoratorklasse: Käse als Pizzaauflage.

Salami als Auflage kostet 1,50 Euro. Eine Pizza mit Salami ist nicht vegetarisch, unabhängig vom darunterliegenden Rest der Pizza. Die Methode `isVegetarian` ruft deshalb die ererbte Methode nicht auf, sondern gibt sofort und ohne weitere Rückfrage das Ergebnis `false` zurück. Was immer unter dieser Salami liegt, die ganze Pizza kann wegen dieser Salami nicht vegetarisch sein.

```
public class Salami extends Topping {
    public Salami(Pizza below) {
        super(below);
    }

    public int getPrice() {
        return 150 + super.getPrice();
    }

    public boolean isVegetarian() {
        return false;
    }
}
```

Listing C.8: Konkrete Dekoratorklasse: Salami als Pizzaauflage.

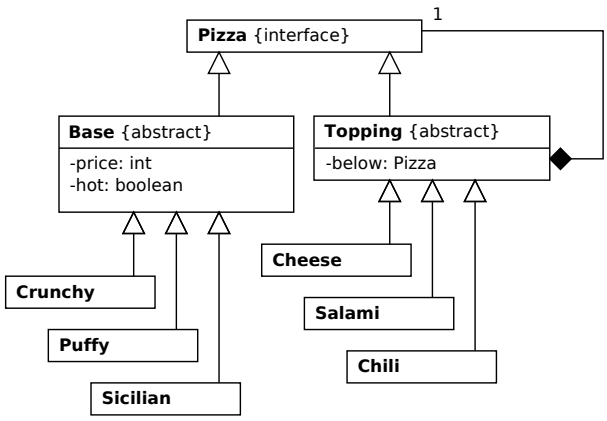
Gewürze sind kostenlos, machen eine Pizza aber scharf.

```
public class Chili extends Topping {
    public Chili(Pizza below) {
        super(below);
    }

    public boolean isHot() {
        return true;
    }
}
```

Listing C.9: Konkrete Dekoratorklasse: Gewürze als Pizzaauflage.

Grundstruktur des Decorator-Patterns

Das folgende Diagramm zeigt die Beziehungen zwischen den Typen. Darin lässt sich die Grundstruktur des Decorator-Patterns erkennen (Seite 73).



Aufbau einer Objektstruktur zur Laufzeit

Eine Anwendung kann aus diesen Typen zur Laufzeit beliebige Pizzas zusammenstellen und deren Eigenschaften abfragen. Das folgende Programm erwartet auf der Kommandozeile eine Reihe von Kürzeln mit jeweils den beiden kleinen Anfangsbuchstaben der gewünschten Bausteine. Als Erstes muss ein Pizzaboden genannt werden, daran anschließend die Auflagen.

```
public class PizzaMain {
    public static void main(String... args) {
        Pizza pizza = null;
        for(String arg: args)
            if(pizza == null)
                switch(arg) {
                    case "Crunchy":
                        pizza = new Crunchy();
                        break;
                    case "Puffy":
                        pizza = new Puffy();
                        break;
                    case "Sicilian":
                        pizza = new Sicilian();
                        break;
                }
            else
                switch(arg) {
                    case "Cheese":
                        pizza = new Cheese(pizza);
                        break;
                    case "Salami":
                        pizza = new Salami(pizza);
                        break;
                    case "Chili":
                        pizza = new Chili(pizza);
                        break;
                }
        System.out.printf("Your pizza:%nprice: %d%nvegetarian: %b%nhot: %b%n",
                          pizza.getPriceVerbose(""),
```

```
                            pizza.isVegetarian(),
                            pizza.isHot());
    }
}
```

Listing C.10: Programm, das eine Pizza nach Kommandozeilenargumenten zusammensetzt und ihre Eigenschaften ausgibt.

Beim Start des Programms können beliebige Pizzas zusammengestellt werden. Im folgenden Beispiel wird ein Pizzaboden der Art „Crunchy“ mit zweimal Käse, einmal Salami und einmal Gewürzen belegt. Das Ergebnis ist eine Pizza mit einem Gesamtpreis von 6,50 Euro, die nicht vegetarisch, aber scharf ist:

```
$ java PizzaMain Crunchy Cheese Cheese Salami Chili
Your pizza:
    price: 650
    vegetarian: false
    hot: true
```

Rekursive Methodenaufrufe

Die folgende Ausgabe macht die Aufrufe der `getPrice`-Methoden im obigen Beispiel sichtbar. In spitzen Klammern ist jeweils die Objektreferenz angegeben, anhand derer gleiche und verschiedene Objekte identifiziert werden können.[2] Die Aufrufkette beginnt mit dem vordersten Dekorator, einem `Chili`-Objekt. `Chili` definiert keine eigene `getPrice`-Methode, sondern erbt sie von der Basisklasse. Das `Topping`-Objekt, dessen `getPrice`-Aufruf als Erstes protokolliert wird, delegiert den Aufruf an den nächsten Dekorator, das `Salami`-Objekt, das wiederum die Basisklassenmethode aufruft. Die Aufrufkette endet beim `Base`-Objekt, das ohne weitere Aufrufe den eigenen Preis, 300, zurückliefert. Bei der Rückkehr addieren die konkreten Auflagen ihren eigenen Preis zum Preis der restlichen Pizza und geben die Summe zurück:

```
$ java PizzaMain Crunchy Cheese Cheese Salami Chili
Topping@385715.getPrice() =>
  Salami@dd23cf.getPrice() =>
    Topping@dd23cf.getPrice() =>
      Cheese@5a25f3.getPrice() =>
        Topping@5a25f3.getPrice() =>
          Cheese@717ef5.getPrice() =>
            Topping@717ef5.getPrice() =>
              Base@1461c98.getPrice() =>
              <= Base@1461c98.getPrice(): 300
            <= Topping@717ef5.getPrice(): 300
```

[2] Der Wert des Codes ist ohne Bedeutung. Er wird von der `toString`-Methode geliefert, die in `Object` definiert ist. Es handelt sich um die hexadezimale Darstellung des Hashcodes, der auf der Speicheradresse des Objekts beruht.

```
          <= Cheese@717ef5.getPrice(): 400
        <= Topping@5a25f3.getPrice(): 400
      <= Cheese@5a25f3.getPrice(): 500
    <= Topping@dd23cf.getPrice(): 500
  <= Salami@dd23cf.getPrice(): 650
<= Topping@385715.getPrice(): 650
```

Den Bausteinen des Decorator-Patterns sind die folgenden Typen der Pizza-Implementierung zugeordnet:

AbstractComponent
: `Pizza`

ConcreteComponent
: `Base`, `Crunchy`, `Puffy`, `Sicilian`

AbstractDecorator
: `Topping`

ConcreteDecorator
: `Cheese`, `Salami`, `Chili`

Grenzen des Musters

So elegante Lösungen das Decorator-Pattern für bestimmte Probleme auch liefert, so hat es doch auch Grenzen:

- Stellen Sie sich vor, eine Pizza gilt erst dann als „scharf“, wenn *zweimal* Chili als Auflage verwendet wird. Diese unscheinbare Änderung macht einen Zähler für Chili nötig, der im Interface verankert werden muss. Der Zähler wird damit öffentlich sichtbar, obwohl ihn der Anwender nicht verlangt hat und auch nicht nutzen sollte.[3]
- Als zusätzliche Eigenschaft soll die Größe von Pizzas berücksichtigt werden. Auf den Gesamtpreis einer Pizza wirkt sich die Größe als Faktor aus, mit dem der Grundpreis am Ende multipliziert wird. Es stellt sich die Frage, an welcher Stelle diese Multiplikation stattfinden soll. Letztlich muss jede Komponente für sich die Multiplikation ausführen, weil keine einzelne Komponente die Gesamtkonstruktion kennt.

[3] Man könnte eine entsprechende Methode vielleicht als `protected` definieren und damit die Sichtbarkeit einschränken. Allerdings ist das einerseits in einem Interface nicht möglich. Andererseits hat jede abgeleitete Klasse Zugriff auf `protected`-Elemente. Das gilt insbesondere für abgeleitete Klassen in beliebigen fremden Packages. Der Zugriffsschutz `protected` ist so gesehen kaum wirksamer als `public`.

Anhang

D Java-8-Entwicklerversion

Dieser Anhang ist mit Erscheinen einer finalen Version von Java 8 obsolet.

Java 8 erscheint nach Stand der Planung vom Winter 2011/2012 im Sommer 2013. Dieser Anhang gibt Hinweise, wie Sie vor dem Erscheinen einer finalen Version trotzdem einige Erweiterungen von Java 8 praktisch erproben können. Dieses Buch diskutiert die folgenden Neuerungen:

- Lambda-Ausdrücke (Abschnitt 5.5)
- Default-Methoden (Abschnitt 5.6)

Jedes dieser Sprachmittel kann mit passenden Maßnahmen „probegefahren“ werden. Bitte behalten Sie aber im Auge, dass diese Testversionen weder vollständig noch stabil genug für den Alltagseinsatz sind. Rechnen Sie damit, dass mancher Quelltext den Compiler zum Absturz bringt oder dass ein formal korrektes Programm mit sonderbaren Laufzeitfehlern abbricht.

Sie haben sicher bereits eine Version des JDK (*Java Development Kit*) installiert. Diese Installation wird nicht verändert und nimmt keinen Schaden. Alle hier gezeigten Maßnahmen sind temporär. Wenn Sie die heruntergeladenen und ausgepackten Daten wieder löschen, bleiben keine Spuren.

D.1 Lambda-Ausdrücke

Auf der Webseite

```
http://jdk8.java.net/lambda
```

steht eine Entwicklerversion von Java SE 8 mit Unterstützung von Lambda-Ausdrücken zur Verfügung.[1]

Laden Sie die passende Datei für Ihr System herunter und packen Sie sie aus. Auf einem Linux-Rechner kann das folgendermaßen aussehen:

```
$ wget http://download.java.net/lambda/b1314/linux-i586/lambda-8-b1314-linux-i586-10_nov_
$ tar xf lambda-8-b1314-linux-i586-10_nov_2011.tar.gz
```

Gelegentlich stellen die Entwickler von Java 8 eine neue Version bereit. Der konkrete Dateiname ändert sich dabei. Um mit der neuen Java-Version zu arbeiten, definieren Sie zwei Umgebungsvariablen.

- `JAVA_HOME` verweist auf das Directory, in dem Sie den Download ausgepackt haben. Sie vermeiden viele Probleme, wenn Sie einen absoluten Pfad verwenden
- `PATH` erweitert den Suchpfad für Programme, hier `javac` und `java`, um das Subdirectory `bin` im Directory `JAVA_HOME`.

Leider werden Umgebungsvariablen nicht auf allen Systemen gleich gesetzt. Hier einige Varianten, wobei die Lage des Directory `jdk1.8.0` nur als Beispiel dient und bei Ihnen anders sein wird:

Windows:

```
C:> set JAVA_HOME=c:\temp\jdk1.8.0
C:> set PATH=%JAVA_HOME%\bin;%PATH%
```

Unix, *bash*:

```
$ export JAVA_HOME=/var/tmp/jdk1.8.0
$ export PATH=$(pwd)/bin:$PATH
```

Unix, *tcsh*:

```
> set JAVA_HOME=/var/tmp/jdk1.8.0
> set PATH=‘pwd‘/bin:$PATH
```

Test

Nach diesen Maßnahmen sollten sich Compiler und JVM mit Version 8 melden:

[1] Es gibt auf `http://jdk8.java.net/download.html` noch eine andere Fassung des JDK 8, in der *keine* Lambda-Ausdrücke implementiert sind. Diese Version eignet sich nicht.

```
$ javac -version
javac 1.8.0-ea
$ java -version
openjdk version "1.8.0-ea"
OpenJDK Runtime Environment (build 1.8.0-ea-b1314)
OpenJDK Server VM (build 23.0-b04, mixed mode)
```

Das folgende Programm erlaubt einen ersten Schnelltest von Lambda-Ausdrücken:

```
interface Value<T> {
    T calculate();
}

public class HelloLambda {
    public static void main(String... args) {
        Value<Integer> v = () -> 42;
        System.out.println(v.calculate());
    }
}
```

Listing D.1: Minimaler Test von Lambda-Ausdrücken.

Es sollte fehlerfrei übersetzt werden und ablaufen:

```
$ javac HelloLambda.java
$ java HelloLambda
42
```

D.2 Default-Methoden

Der Compiler der Entwicklerversion von Java SE 8 (siehe vorhergehender Abschnitt) übersetzt Default-Methoden, allerdings kommt die JVM nicht damit zurecht.

Eine Erweiterungen in Form eines **Java-Agenten** modifiziert die JVM so, dass sie auch Default-Implementierungen findet. Java-Agenten sind selbst Java-Programme, die die JVM instrumentieren. Der hier benötigte Java-Agent erweitert die Methodenauflösung um Default-Methoden. Er steht auf

```
http://jsr335-lambda.googlecode.com
```

in einem Subversion-Repository zum Download zur Verfügung. Holen Sie sich eine Arbeitskopie des Repository auf Ihren Rechner:

```
$ svn checkout http://jsr335-lambda.googlecode.com/svn/trunk/ jsr335-lambda
```

Der Quelltext des Java-Agenten liegt im Subdirectory `jsr335-lambda/agent`. Übersetzen Sie den Agenten dort:

```
$ cd jsr335-lambda/agent
$ ant
```

Anschließend finden Sie im Subdirectory `lib` die neue Jar-Datei `jsr335-agent.jar`. Speichern Sie den absoluten Pfadnamen dieser Jar-Datei in einer Umgebungsvariablen, beispielsweise `AGENT`:[2]

```
$ export AGENT=$(pwd)/lib/jsr335-agent.jar
```

Rufen Sie ab jetzt die JVM mit einem Schalter auf, der den Java-Agenten einbindet:

```
$ java -javaagent:$AGENT
```

Beachten Sie, dass im gleichen Directory wie `jsr335-agent.jar` auch die Jar-Datei `asm-all-4.0.jar` liegen muss. Sie ist Teil des Downloads und muss nicht erst generiert werden.

Test

Übersetzen Sie das folgende Programm:

```
interface Default {
    void call(int i) default {
        System.out.println(i);
    }
}

public class HelloDefault implements Default {
    public static void main(String... args) {
        new HelloDefault().call(42);
    }
}
```

Listing D.2: Minimaler Test von Default-Methoden.

2 Der konkrete Name der Umgebungsvariablen spielt keine Rolle.

Mit dem Agenten sollte das Programm fehlerfrei arbeiten:

```
$ javac HelloDefault.java
$ java -javaagent:$AGENT HelloDefault
42
```

Literaturverzeichnis

[1] Extensible Markup Language (XML) 1.0 (Fifth Edition), 26.11.2008
`http://www.w3.org/TR/REC-xml`

[2] Namespaces in XML 1.0 (Third Edition), 08.12.2009
`http://www.w3.org/TR/REC-xml`

[3] Joshua Bloch: Effective Java, A Programming Language Guide
Addison-Wesley Longman, Amsterdam, 2. Auflage 2008

[4] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: *The Java Language Specification*, 3. Auflage 2005
`http://java.sun.com/docs/books/jls`, 3. Auflage 2005

Index