David J. Barnes • Michael Kölling

Java lernen mit BlueJ

Objects first – Eine Einführung in Java

6., aktualisierte Auflage

Pearson

EXTRAS ONLINE

Übung 5.14 Schreiben Sie die Methode **sichtungenAusgebenVon** in Ihrem Projekt so wie oben beschrieben neu.

Übung 5.15 Schreiben Sie eine Methode, um die Anzahlen aller Sichtungen einer bestimmten Tierart auszugeben. Ihre Methode sollte die **map**-Operation als Teil der Pipeline einsetzen.

Übung 5.16 Wenn eine Pipeline eine **filter**- und eine **map**-Operation enthält, ist die Reihenfolge der Operationen ausschlaggebend für das Endresultat? Begründen Sie Ihre Antwort.

Übung 5.17 Schreiben Sie die Methode **gefaehrdedAusgeben** in Ihrem Projekt neu, sodass sie nun Streams einsetzt, und testen Sie sie. (Um diese Methode zu testen, kann es am einfachsten sein, eine Testmethode zu schreiben, die eine **ArrayList** aus Tiernamen erzeugt und die **gefaehrdedAusgeben**-Methode damit aufruft.)

Übung 5.18 *Zusatzaufgabe*. Die Methode **sichtungenAusgebenVon** von **Tiermonitor** enthält die folgende terminale Operation:

```
forEach(details -> System.out.println(details));
```

Ersetzen Sie diese durch

```
forEach(System.out::println);
```

Achten Sie darauf, die korrekte Syntax zu benutzen. Verändert dies die Funktion der Methode? Was können Sie aus Ihrer Antwort ableiten? Wenn Sie sich nicht sicher sind, versuchen Sie herauszufinden, was **::** bei der Verwendung von Lambda-Ausdrücken in Java bedeutet. Wir werden uns dies in den Fortgeschrittenenabschnitten von Kapitel 6 ansehen.

5.5.5 Die Methode reduce

Die intermediären Operationen, die wir bisher betrachtet haben, bekommen einen Stream als Eingabe und geben einen Stream als Ausgabe zurück. Manchmal benötigen wir jedoch eine Operation, die ihren Eingabe-Stream auf ein einziges Objekt oder Wert „kollabieren" lässt, und diese Rolle kommt der **reduce**-Methode als terminaler Operation zu.

Der vollständige Code für das obige Beispiel – alle Tiere eines gegebenen Typs auswählen, diese dann auf die Anzahl der Tiere in jeder Sichtung abbilden und abschließend alle Zahlen addieren – lautet wie folgt:

```
public int gibAnzahl(String tier)
{
   return sichtungen.stream()
      .filter(sichtung> tier.equals(sichtung.gibTier()))
      .map(sichtung -> sichtung.gibAnzahl())
      .reduce(0, (total, anzahl) -> total + anzahl);
}
```

Wir sehen, dass die Parameter für die **reduce**-Methode ein wenig aufwendiger aussehen als für die **filter**- und **map**-Methoden. Dafür gibt es zwei Gründe:

- Die **reduce**-Methode hat zwei Parameter. Der erste in unserem Beispiel ist 0, der zweite ist ein Lambda-Ausdruck.
- Der hier verwendete Lambda-Ausdruck hat selbst wiederum zwei Parameter, **total** und **anzahl**.

Um die Funktionsweise zu verstehen, ist es ganz hilfreich, wenn wir uns ansehen, wie wir den Code schreiben müssten, wollten wir die Anzahlen auf die traditionelle Art und Weise addieren. Die Schleife sähe dann so aus:

```
public int gibAnzahl(String tier)
{
   int total =0;
   for(Sichtung sichtung : sichtungen) {
      if(tier.equals(sichtung.gibTier())) {
         total = total + sichtung.gibAnzahl())) {
      }
   }
   return total;
}
```

Die Berechnung der Summe umfasst die folgenden Schritte:

- Deklariere eine Variable für die Aufnahme des Endwerts und gib ihr den Anfangswert 0.
- Iteriere über der Liste und stelle fest, welche Sichtungsdatensätze sich auf die Tierart beziehen, an der wir interessiert sind (der Filter-Schritt).
- Erhalte die Anzahl der identifizierten Sichtungen (der Map-Schritt).
- Addiere die Anzahl zur Variablen.
- Gib die Summe zurück, die in der Variablen im Verlauf der Iteration summiert wurde.

Der zentrale Punkt, der relevant für das Programmieren einer Stream-basierten Version dieses Prozesses ist, ist zu erkennen, dass die Variable **total** als eine Art „Akkumulator" fungiert: Jedes Mal, wenn eine relevante Anzahl identifiziert wird, wird diese zu dem Wert addiert, der bisher in **total** angelaufen ist. Dieser neue Wert wird beim nächsten Schleifendurchlauf verwendet. Damit dies funktioniert, muss **total** offensichtlich einen Anfangswert von 0 haben, auf den die erste Anzahl addiert wird.

Die **reduce**-Methode hat zwei Parameter: einen Startwert und einen Lambda-Ausdruck. Formal wird der Startwert *Identität* genannt. Es ist der Wert, mit dem unsere laufende Variable **total** initialisiert wird. In unserem Code haben wir 0 für diesen Parameter übergeben.

Der zweite Parameter ist ein Lambda-Ausdruck mit zwei Parametern: einer für die laufende Variable **total** und einer für das aktuelle Element des Streams. Der Lambda-Ausdruck in unserem Code lautet:

```
(total, anzahl) -> total + anzahl
```

Der Rumpf des Lambda-Ausdrucks sollte den Wert liefern, der durch die Kombination der laufenden Variablen **total** mit dem Element resultiert. In unserem Beispiel heißt „kombinieren“ einfach nur addieren. Die Anwendung dieser **reduce**-Methode bewirkt dann, dass **total** mit null initialisiert wird, jedes Element des Streams hinzuaddiert wird und zum Schluss **total** zurückgegeben wird.

Da es in unserem Code keine explizite Schleife mehr gibt, ist er anfangs vielleicht ein wenig schwieriger zu verstehen, doch insgesamt werden dieselben Elemente wie in der Version mit der **forEach**-Schleife verwendet.

Übung 5.19 Schreiben Sie Ihre **gibAnzahl**-Methode neu, indem Sie wie hier gezeigt Streams verwenden.

Übung 5.20 Fügen Sie eine Methode zu **Tiermonitor** hinzu, die drei Parameter hat, **tier**, **melderID** und **tagID**, und eine Anzahl zurückgibt, wie viele Sichtungen einer gegebenen Tierart von diesem Melder an einem bestimmten Tag gemacht wurden.

Übung 5.21 Fügen Sie **Tiermonitor** eine Methode hinzu, die zwei Parameter bekommt – **melderID** und **tagID** – und einen **String** zurückliefert, der die Namen der gesichteten Tiere zurückgibt, die vom Melder am gegebenen Tag gesehen wurden. Sie sollten nur Tiere aufnehmen, deren Sichtungszählung größer als null ist, doch Sie müssen sich nicht um doppelt vorkommende Namen kümmern, falls es mehrere nicht leere Sichtungsdatensätze von einem bestimmten Tier gibt. *Hinweis:* Die Verwendung von **reduce** mit **String**-Elementen und einem **String**-Ergebnis ist im Prinzip ganz ähnlich dem Vorgehen mit ganzen Zahlen. Entscheiden Sie sich für die richtige Identität und formulieren Sie einen Lambda-Ausdruck mit zwei Parametern, der die laufende „Summe“ mit dem nächsten Element des Streams kombiniert.

5.5.6 Elemente aus einer Sammlung entfernen

Wir haben gesagt, dass die **filter**-Methode die zugrunde liegende Sammlung, von der der Stream stammt, eigentlich nicht verändert. Prädikat-Lambda-Ausdrücke machen es relativ leicht, alle Elemente aus der Sammlung zu entfernen, die einer bestimmten Bedingung genügen. Nehmen wir zum Beispiel an, wir möchten aus der **sichtungen**-Liste alle **Sichtung**-Datensätze löschen, deren Anzahl null ist. Der folgende Code verwendet die **removeIf**-Methode der Sammlung:

```
/**
 * Entferne aus der sichtungen-Liste alle Datensätze
 * mit Anzahl null.
 * @param melder  die ID des Melders
 */
public void entferneNullAnzahl()
{
   sichtungen.removeIf(sichtung -> sichtung.gibAnzahl() == 0);
}
```

Wieder einmal ist die Iteration implizit. Die `removeIf`-Methode übergibt jedes Element der Liste nacheinander dem Prädikat-Lambda-Ausdruck, entfernt dann alle Elemente, für die der Lambda-Ausdruck `true` liefert. Beachten Sie, dass dies eine Methode der Sammlung ist (nicht eines Streams) und hiermit die ursprüngliche Sammlung verändert wird.

Übung 5.22 Schreiben Sie die Methode `entferneNullAnzahl` neu, indem Sie die `removeIf`-Methode verwenden, wie oben gezeigt.

Übung 5.23 Schreiben Sie eine Methode `entferneMelder`, der alle Datensätze entfernt, die von einem gegebenen Melder stammen.

5.5.7 Weitere Stream-Methoden

Wir haben gesagt, dass viele der häufig vorkommenden Aufgaben mit diesen drei Operationsarten – *filter*, *map* und *reduce* – bewerkstelligt werden können. In der Praxis bietet Java viele weitere Stream-Methoden, einige davon werden wir später noch kennenlernen. Die meisten sind allerdings lediglich Variationen der drei hier vorgestellten Operationen, auch wenn sie andere Namen haben.

Die Klasse `Stream` hat beispielsweise Methoden namens `count`, `findFirst` und `max` – alles Varianten einer *reduce*-Operation – sowie Methoden namens `limit` und `skip`, die Beispiele einer *filter*-Operation sind.

Übung 5.24 Finden Sie die API-Dokumentation für `Stream` im Paket `java.util.stream` und sehen Sie sich die Methoden `count`, `findFirst` und `max` an. Diese können für jeden Stream aufgerufen werden.

Übung 5.25 Schreiben Sie eine Methode in der Klasse `Tiermonitor`, die einen einzelnen Parameter, `melderID`, bekommt und eine Anzahl zurückgibt, wie viele Sichtungsdatensätze von diesem Melder gemacht wurden. Verwenden Sie dazu die `Stream`-Methode `count`.

Übung 5.26 Schreiben Sie eine Methode, die den Namen einer Tierart als Parameter hat und die größte Anzahl für dieses Tier in einem einzelnen `Sichtung`-Datensatz zurückliefert.

Übung 5.27 Schreiben Sie eine Methode, die den Namen einer Tierart und eine `melderID` als Parameter hat und das erste `Sichtung`-Objekt liefert, das in der `sichtungen`-Sammlung für diese Kombination gespeichert ist.

Übung 5.28 Lesen Sie sich die Methoden `limit` und `skip` von `Stream` durch und erfinden Sie selbst einige Methoden, um sie zu verwenden.

Zusammenfassung

In diesem Kapitel haben wir einige neue Konzepte eingeführt, die relativ fortgeschritten für dieses Stadium des Buchs sind, aber die sehr eng mit den Konzepten verwandt sind, die wir in Kapitel 4 behandelt haben. Wir haben uns einen *funktionalen Stil* zur Verarbeitung von Datenströmen angesehen. Bei diesem Stil schreiben wir keine Schleifen, um die Elemente in einer Sammlung zu verarbeiten. Stattdessen wenden wir eine *Pipeline* von Operationen auf einen Stream an, der aus einer Sammlung abgeleitet ist, um die Folge in die gewünschte Form zu transformieren. Jede Operation in der Pipeline definiert, wie ein einzelnes Element der Folge behandelt werden soll. Die häufigsten Transformationen werden mittels der Methoden `filter`, `map` und `reduce` durchgeführt.

Operationen in einer Pipeline haben häufig *Lambda-Ausdrücke* als Parameter, um zu spezifizieren, was mit jedem Element der Folge getan werden soll. Ein Lambda-Ausdruck ist eine anonyme Funktion, die keinen oder mehrere Parameter sowie einen Codeblock hat, der dieselbe Rolle wie ein Methodenrumpf spielt.

Wenn Sie sich das Projekt *Tiermonitoring-v1* ansehen, mit dem Sie gearbeitet haben, dann haben Sie vielleicht bemerkt, dass es noch zwei Methoden gibt, die `forEach`-Schleifen verwenden und die wir bisher noch nicht für die Verwendung von Streams umgeschrieben haben. Dies sind Methoden, die neue Sammlungen als Methodenergebnis zurückgeben; dieses Thema – eine neue Sammlung aus einem Stream zu erzeugen – haben wir noch nicht behandelt. Wir kommen darauf in Kapitel 6 zurück, wo wir sehen werden, wie eine neue Sammlung aus der transformierten Folge gewonnen werden kann.

Zentrale Konzepte in diesem Kapitel: Funktionale Programmierung, Lambda-Ausdruck, Stream, Pipeline, `filter`, `map`, `reduce`

Zusammenfassung der Konzepte

- **Lambda-Ausdruck** Ein Lambda-Ausdruck ist ein Codesegment, das für eine spätere Ausführung gespeichert werden kann.
- **Funktionaler Stil** Im funktionalen Stil der Verarbeitung von Sammlungen entnehmen wir nicht jedes Element einzeln, um damit zu arbeiten. Stattdessen übergeben wir der Sammlung ein Codesegment, das auf jedes Element angewandt wird.
- **Streams** Streams vereinheitlichen die Verarbeitung von Elementen einer Sammlung und anderer Datenmengen. Ein Stream bietet nützliche Methoden, um diese Datenmengen zu manipulieren.
- **filter** Wir können eine `filter`-Operation auf einen Stream anwenden, um nur bestimmte Elemente auszuwählen.
- **map** Wir können einen Stream mithilfe der `map`-Operation auf einen anderen Stream abbilden, wobei jedes Element des ursprünglichen Streams durch ein neues Element ersetzt wird, das aus dem Original abgeleitet ist.
- **reduce** Wir können einen Stream mithilfe der `reduce`-Operation reduzieren, das heißt, wir wenden eine Funktion an, die einen ganzen Stream entgegennimmt und einen einzelnen Ergebniswert zurückgibt.
- **Pipeline** Eine Pipeline ist die Kombination von zwei oder mehreren Stream-Funktionen in einer Kette, wobei jede Funktion der Reihe nach angewendet wird.

Übung 5.29 Erstellen Sie eine Kopie des Projekts *Musiksammlung-v5* aus Kapitel 4. Schreiben Sie die Methoden `alleTracksAusgeben` und `bestimmteTracksAusgeben` der Klasse `MusikSammlung` neu, um Streams und Lambda-Ausdrücke zu verwenden.

KAPITEL

6

Bibliotheksklassen nutzen

Lernziele

Zentrale Konzepte in diesem Kapitel: Mit Bibliotheksklassen umgehen, Dokumentation lesen, Dokumentation schreiben

Java-Konstrukte in diesem Kapitel: `String`, `Random`, `HashMap`, `HashSet`, `Iterator`, `static`, `final`, Autoboxing, Wrapper-Klassen

In Kapitel 4 haben wir die Klasse `ArrayList` aus der Java-Klassenbibliothek vorgestellt. Wir haben gesehen, dass wir mit dieser Klasse Dinge tun können, die sonst nur recht schwierig umzusetzen wären (in unserem Beispiel eine beliebige Anzahl von Objekten speichern).

Dies war nur ein Beispiel für eine nützliche Klasse aus der Java-Bibliothek. Die Bibliothek besteht aus Tausenden von Klassen, von denen viele sehr nützlich für Ihre Arbeit sein können, während Sie vermutlich viele andere niemals benötigen werden.

Für einen guten Java-Programmierer ist es wichtig, kompetent mit der Java-Bibliothek arbeiten zu können. Vor allem die Fähigkeit, die richtigen Klassen aus ihr auszuwählen, ist unerlässlich. Sobald Sie mit der Bibliothek einigermaßen vertraut sind, werden Sie merken, dass Sie vieles sehr viel einfacher und schneller mithilfe der Bibliotheksklassen erledigen können als ohne sie. Der Umgang mit diesen Klassen ist der Schwerpunkt in diesem Kapitel.

Bei den Elementen in der Bibliothek handelt es sich nicht um einen Satz von beliebigen Klassen, die in keinem Zusammenhang stehen und die wir alle einzeln lernen müssen, sondern sie werden oft nach bestimmte Kriterien zusammengestellt, die auf Gemeinsamkeiten beruhen. Auch hier haben wir es wieder mit dem Konzept der Abstraktion zu tun, die uns hilft, eine Fülle an Informationen zu verarbeiten. Zu den wichtigsten Teilen der Bibliothek gehören die Sammlungen, zu denen auch die `ArrayList` gehört. Wir werden in diesem Kapitel weitere Sammlungen besprechen und feststellen, dass sie viele Attribute gemeinsam haben, sodass wir oft von den individuellen Details einer speziellen Sammlung abstrahieren und über Sammlungsklassen im Allgemeinen sprechen können.

Wir werden neue Sammlungsklassen sowie einige andere nützliche Bibliotheksklassen vorstellen und diskutieren. Im Verlauf dieses Kapitels werden wir uns mit der Erstellung einer einzelnen Anwendung beschäftigen (einem System für einen

technischen Kundendienst), die viele verschiedene Bibliotheksklassen benutzt. Eine komplette Implementierung mit allen Entwurfsideen und Quelltexten, die wir hier betrachten, samt einiger Zwischenversionen finden Sie in den Buchprojekten. Obwohl Sie dadurch die komplette Lösung untersuchen können, schlagen wir vor, dass Sie die Übungen in diesem Kapitel Schritt für Schritt nachvollziehen. Nachdem wir einen kurzen Blick auf das komplette System geworfen haben, werden wir von einer sehr einfachen Basisversion ausgehen und dann nach und nach die komplette Lösung entwerfen und implementieren.

Die Anwendung benutzt mehrere Bibliotheksklassen und viele für Sie neue Techniken, die alle für sich einige Aufmerksamkeit erfordern: Hash-Abbildungen, Mengen, Zeichenkettenzerlegung und die weitere Verwendung von Zufallszahlen. Sie sollten sich darauf einstellen, dass Sie dieses Kapitel nicht innerhalb eines Tages lesen und verstehen können, sondern dass es etliche Abschnitte enthält, für die Sie einige Tage intensiven Studiums investieren müssen. Dafür werden Sie am Ende, wenn Sie alle Übungen programmiert haben, eine Vielzahl wichtiger Konzepte gelernt haben.

6.1 Die Dokumentation der Bibliotheksklassen

Konzept

Die **Standardklassenbibliothek** von Java enthält viele Klassen, die sehr nützlich sind. Es ist wichtig zu wissen, wie man die Bibliothek benutzen kann.

Die Standardklassenbibliothek von Java ist riesig. Sie besteht aus Tausenden von Klassen, jede mit vielen Methoden, mit oder ohne Parameter und mit oder ohne Ergebnistypen. Es ist unmöglich, sie alle mitsamt den Details ihrer Benutzung zu lernen. Ein guter Java-Programmierer sollte stattdessen

- einige der wichtigsten Klassen der Bibliothek und ihre Methoden namentlich kennen (`ArrayList` ist eine dieser wichtigen) und
- wissen, wie er sich die anderen Klassen (mitsamt den Details ihrer Methoden und Parameter) anlesen kann.

In diesem Kapitel werden wir einige der wichtigsten Klassen der Bibliothek vorstellen, weitere werden in späteren Kapiteln behandelt werden. Aber viel wichtiger ist, dass Sie auch lernen, wie Sie sich in der Bibliothek allein zurechtfinden können. Dies wird Ihnen ermöglichen, sehr viel interessantere Programme zu schreiben. Glücklicherweise ist die Java-Bibliothek recht gut dokumentiert. Diese Dokumentation liegt im HTML-Format vor, sodass sie mit einem Webbrowser betrachtet werden kann. Wir werden dies für unsere Erkundung der Bibliotheksklassen ausnutzen.

Lesen und Verstehen der Dokumentation ist der erste Teil unserer Einführung in die Bibliotheksklassen. Wir gehen dann noch einen Schritt weiter, indem wir auch diskutieren, wie Sie eigene Klassen so aufbereiten können, dass andere sie genauso benutzen können, als wären sie Standardbibliotheksklassen. Dies ist für die reale Softwareentwicklung sehr wichtig, bei der über einen längeren Zeitraum in Teams an großen Projekten oder in der Softwarewartung gearbeitet wird.

Bei der Benutzung der Klasse `ArrayList` ist Ihnen möglicherweise aufgefallen, dass wir sie einsetzen konnten, ohne einen Blick auf ihren Quelltext geworfen zu haben. Wir haben nicht überprüft, wie sie implementiert ist. Dies war nicht notwendig, um ihre Dienste in Anspruch zu nehmen. Wir kannten lediglich den Namen der Klasse, die Namen der Methoden mit ihren Parametern und Ergebnistypen und wussten,

was die Methoden tun. Wir haben uns nicht dafür interessiert, wie sie ihre Aufgaben erfüllen. Dies ist typisch für die Benutzung von Bibliotheksklassen.

Dies gilt in gleicher Weise für andere Klassen in großen Softwareprojekten. Typischerweise kooperieren mehrere Entwickler in einem Projekt, indem sie an unterschiedlichen Teilen des Systems arbeiten. Jeder Entwickler sollte sich auf seinen Bereich konzentrieren und nicht die Details der anderen Teile verstehen müssen (wir haben dies bereits in Abschnitt 3.2 diskutiert, als wir über Abstraktion und Modularisierung gesprochen haben). Letztlich sollte jeder Entwickler die Klassen der anderen Teammitglieder so benutzen können, als ob sie Bibliotheksklassen wären: also ihre Dienste in Anspruch nehmen, ohne die Details der Realisierung kennen zu müssen.

Damit dies funktioniert, muss jedes Teammitglied seine Klassen in der Art dokumentieren, wie die Standardbibliothek dokumentiert ist. Diese Dokumentation ermöglicht es dann, die Klasse zu benutzen, ohne den Quelltext zu lesen. Auch dieses Thema werden wir in diesem Kapitel besprechen.

6.2 Das Kundendienstsystem

Wie immer werden wir diese Themen anhand eines Beispiels diskutieren. Diesmal verwenden wir das *Kundendienstsystem*. Sie finden das Projekt im Begleitmaterial unter dem Namen *Technischer-Kundendienst1*.

Das *Kundendienstsystem* ist ein Programm, das den technischen Kundendienst für die Kunden der fiktiven Softwarefirma SeltsamSoft abwickeln soll. Noch vor einiger Zeit hatte SeltsamSoft eine Abteilung für den technischen Kundendienst, in der Mitarbeiter an Telefonen saßen. Kunden konnten dort anrufen, um Rat und Hilfe bei technischen Problemen mit den Softwareprodukten von SeltsamSoft zu bekommen. In letzter Zeit ist das Geschäft allerdings nicht mehr sehr gut gelaufen und SeltsamSoft hat aus Kostengründen beschlossen, die Abteilung aufzulösen. Sie wollen nun ein *Kundendienstsystem* entwickeln, das den Eindruck vermitteln soll, dass noch immer Kundendienst angeboten wird. Das System soll die Antworten simulieren, die ein Mitarbeiter im technischen Kundendienst geben könnte. Die Kunden können mit dem System online kommunizieren.

6.2.1 Das Kundendienstsystem erkunden

Übung 6.1 Öffnen und starten Sie das Projekt *Technischer-Kundendienst-komplett*. Sie starten es, indem Sie ein Objekt der Klasse `Kundendienstsystem` erzeugen und seine `starten`-Methode aufrufen. Geben Sie einige Fragen zu Problemen ein, die Sie mit Ihrer Software haben könnten, um das System auszuprobieren. Beobachten Sie, wie es sich verhält. Tippen Sie „ade“ (es ist ein schwäbisches System!), wenn Sie fertig sind. Sie brauchen an dieser Stelle noch nicht den Quelltext zu untersuchen. Dieses Projekt enthält die komplette Lösung, wie wir sie am Ende dieses Kapitels erstellt haben wollen. Der Zweck dieser Übung ist lediglich, dass Sie einen Eindruck vom Zielsystem bekommen.

Eliza

Die Idee für das Kundendienstsystem stammt von dem bahnbrechenden Programm der Künstlichen Intelligenz, Eliza, das von Joseph Weizenbaum am Massachusetts Institute of Technology in den 1960er Jahren entwickelt wurde. Sie können mehr darüber erfahren, indem Sie im Web nach „Eliza" und „Weizenbaum" suchen.

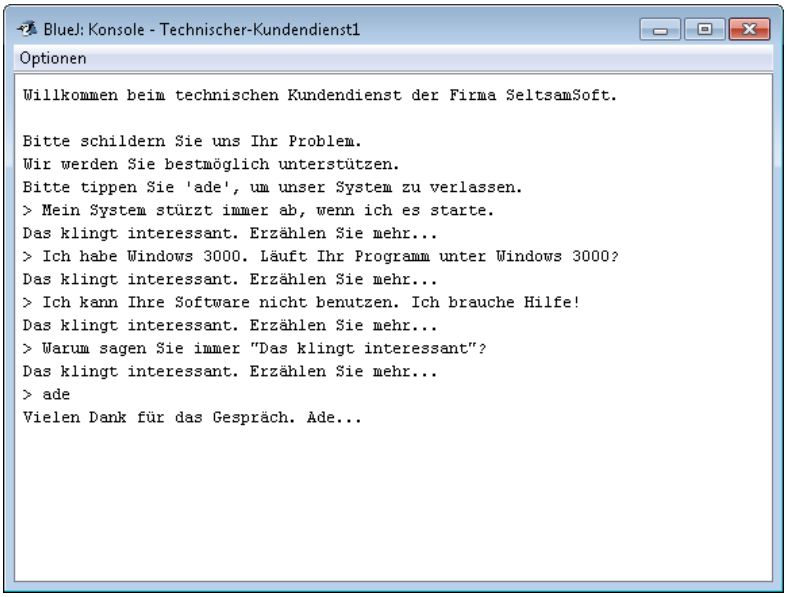
Wir werden nun unsere ausführlichere Untersuchung beginnen, indem wir das Projekt *Technischer-Kundendienst1* öffnen. Es enthält eine erste rudimentäre Implementierung unseres Systems. Wir werden es im Laufe dieses Kapitels ausbauen. Auf diese Weise kommen wir zu einem besseren Verständnis des Gesamtsystems, als wenn wir nur die komplette Lösung betrachten würden.

Aus ***Übung 6.1*** konnten Sie ersehen, dass das System im Kern einen Dialog mit dem Benutzer führt. Der Benutzer kann eine Frage eingeben und das System antwortet. Versuchen Sie nun das Gleiche mit unserem Prototyp des Projekts, *Technischer-Kundendienst1*.

In der kompletten Version schafft es das System, einigermaßen abwechslungsreiche Antworten zu geben. Manchmal scheinen sie sogar sinnvoll zu sein! In dieser ersten Version, die wir weiterentwickeln werden, sind die Reaktionen sehr viel eingeschränkter (Abbildung 6.1). Sie werden schnell feststellen, dass es immer die gleiche Antwort gibt:

Das klingt interessant. Erzählen Sie mehr ...

Abbildung 6.1
Ein erster Dialog mit dem Kundendienstsystem.



Das ist tatsächlich überhaupt nicht interessant und auch nicht sehr überzeugend, wenn eigentlich der Eindruck entstehen soll, dass am anderen Ende der Leitung ein Mitarbeiter des technischen Kundendienstes sitzt. Wir werden bald versuchen, diese Situation zu verbessern. Allerdings sollten wir zuerst etwas genauer untersuchen, was wir bisher vorliegen haben.

Das Projektdiagramm zeigt uns drei Klassen: **`Kundendienstsystem`**, **`Eingabeleser`** und **`Beantworter`** (Abbildung 6.2). **`Kundendienstsystem`** ist die Hauptklasse, die einen **`Eingabeleser`** für Eingaben von der Tastatur benutzt und einen **`Beantworter`**, der Antworten erzeugt.

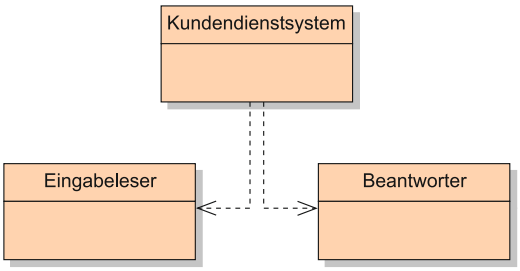


Abbildung 6.2
Das Klassendiagramm von *Technischer-Kundendienst*.

Untersuchen Sie den **`Eingabeleser`** genauer, indem Sie ein Objekt dieser Klasse erzeugen und dann die Objektmethoden betrachten. Sie werden sehen, dass es nur eine Methode anbietet, **`gibEingabe`**, die eine Zeichenkette zurückliefert. Probieren Sie sie aus. Diese Methode lässt auf der Konsole eine Zeile eingeben und liefert Ihre Eingabe als Aufrufergebnis zurück. Wir werden vorläufig nicht im Detail untersuchen, wie dies funktioniert, sondern lediglich festhalten, dass ein **`Eingabeleser`** eine Methode **`gibEingabe`** hat, die eine Zeichenkette zurückliefert.

Machen Sie nun das Gleiche mit der Klasse **`Beantworter`**. Sie werden feststellen, dass sie eine Methode **`generiereAntwort`** hat, die immer die Zeichenkette **`"Das klingt interessant. Erzählen Sie mehr …"`** liefert. Das erklärt, was wir vorher im Dialog beobachtet haben.

Lassen Sie uns nun einen genaueren Blick auf die Klasse **`Kundendienstsystem`** werfen.

6.2.2 Den Quelltext untersuchen

Der komplette Quelltext der Klasse **`Kundendienstsystem`** ist in Listing 6.1 zu sehen. Listing 6.2 zeigt den Quelltext der Klasse **`Beantworter`**.

Copyright