

## 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.

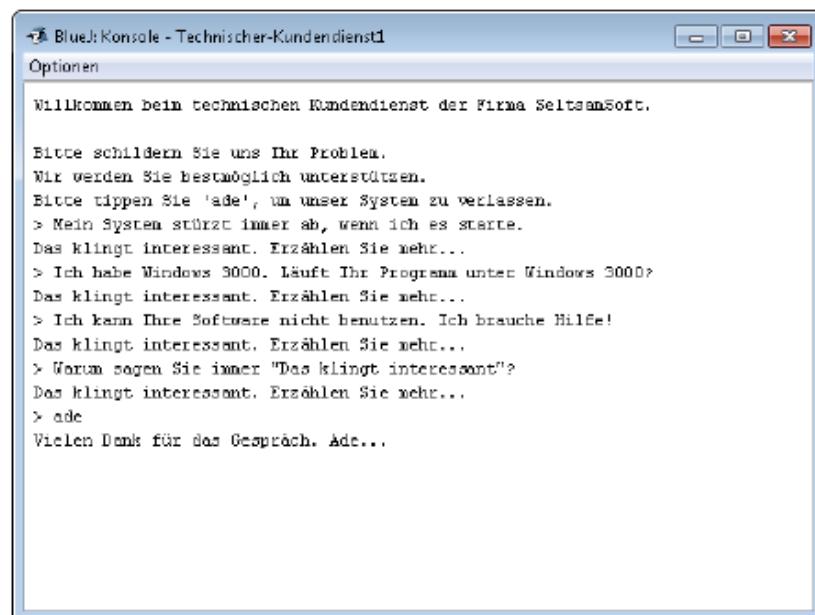
Wir werden nun unsere ausführlichere Untersuchung beginnen, indem wir das Projekt *Technischer-Kundendienst1* öffnen. Es enthält eine erste rudimentäre Implementierung unseres Systems. Wir werden es im Laufe dieses Kapitels ausbauen. Auf diese Weise kommen wir zu einem besseren Verständnis des Gesamt-systems, 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:

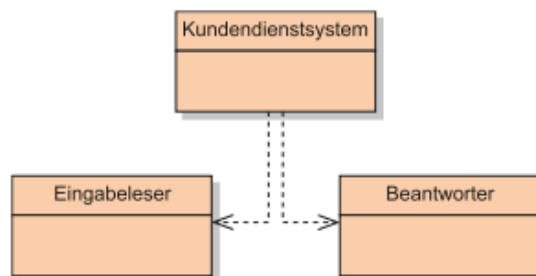
*Das klingt interessant. Erzählen Sie mehr ...*

**Abbildung 6.1**  
Ein erster Dialog mit  
dem Kundendienst-  
system.



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**.

**Listing 6.1**

Der Quelltext der Klasse `Kundendienstsystem`.

```
/*
 * Diese Klasse implementiert ein System für einen technischen
 * Kundendienst. Das System kommuniziert mit dem Benutzer über
 * die Konsole.
 *
 * Diese Klasse benutzt ein Exemplar der Klasse Eingabeleser,
 * um Eingaben vom Benutzer zu lesen, und ein Exemplar der
 * Klasse Beantworter, um Antworten zu generieren.
 * In einer Schleife werden Eingaben eingelesen und Antworten
 * auf diese Eingaben generiert, bis der Benutzer das System
 * verlässt.
 *
 * @author Michael Kölling und David J. Barnes
 * @version 0.1 (2016.02.29)
 */
public class Kundendienstsystem
{
    private Eingabeleser leser;
    private Beantworter beantworter;

    /**
     * Erzeuge ein Kundendienstsystem.
     */
    public Kundendienstsystem()
    {
        leser = new Eingabeleser();
        beantworter = new Beantworter();
    }

    /**
     * Starte das System für den technischen Kundendienst.
     * Es wird ein Begrüßungstext ausgegeben und anschließend
     * ein Dialog mit dem Benutzer geführt, bis der Benutzer
     * den Dialog beendet.
     */
    public void starten()
    {
        boolean fertig = false;

        willkommenstextAusgeben();

        while(!fertig) {
            String eingabe = leser.gibEingabe();

            if(eingabe.startsWith("ade")) {
                fertig = true;
            }
            else {
                String antwort = beantworter.generiereAntwort();
                System.out.println(antwort);
            }
        }
        abschiedstextAusgeben();
    }

    /**
     * Gib einen Willkommenstext auf der Konsole aus.
     */
    private void willkommenstextAusgeben()
    {
        System.out.println("Willkommen beim technischen Kundendienst" +
                           " der Firma SeltSamSoft.");
        System.out.println();
        System.out.println("Bitte schildern Sie uns Ihr Problem.");
        System.out.println("Wir werden Sie bestmöglich unterstützen.");
        System.out.println("Bitte tippen Sie 'ade', um unser System zu" +
                           " verlassen.");
    }

    /**
     * Gib einen Abschiedstext auf der Konsole aus.
     */
    private void abschiedstextAusgeben()
    {
        System.out.println("Vielen Dank für das Gespräch. Ade ...");
    }
}
```

```
/**  
 * Die Klasse Beantworter beschreibt Exemplare, die  
 * automatische Antworten generieren.  
 *  
 * @author Michael Kölling und David J. Barnes  
 * @version 0.1 (2016.02.29)  
 */  
public class Beantworter  
{  
    /**  
     * Erzeuge einen Beantworter - nichts Besonderes zu tun.  
     */  
    public Beantworter()  
    {  
    }  
  
    /**  
     * Erzeuge eine Antwort.  
     * @return einen String, der die Antwort enthält  
     */  
    public String generiereAntwort()  
    {  
        return "Das klingt interessant. Erzählen Sie mehr ...";  
    }  
}
```

**Listing 6.2**

Der Quelltext der Klasse **Beantworter**.

Ein Blick auf Listing 6.2 zeigt uns, dass die Klasse **Beantworter** trivial ist. Sie hat nur eine Methode und diese liefert immer dieselbe Zeichenkette. Das ist etwas, was wir später verbessern werden. Vorläufig konzentrieren wir uns auf die Klasse **Kundendienstsystem**.

**Kundendienstsystem** deklariert zwei Datenfelder, um einen **Eingabeleser** und einen **Beantworter** halten zu können, und initialisiert diese Datenfelder mit entsprechenden Objekten, die im Konstruktor erzeugt werden.

Am Ende hat die Klasse zwei Methoden, **willkommenstextAusgeben** und **abschiedstextAusgeben**. Diese geben einfach kurze Texte aus – eine Begrüßung und einen Abschiedstext.

Der interessanteste Abschnitt im Quelltext ist die Methode in der Mitte: **starten**. Wir werden diese Methode genauer untersuchen.

Zu Anfang der Methode erfolgt ein Aufruf von **willkommenstextAusgeben**, am Ende steht ein Aufruf von **abschiedstextAusgeben**. Diese Aufrufe sorgen für die entsprechenden Ausgaben zu den geeigneten Zeitpunkten. Der Rest der Methode besteht aus der Deklaration einer booleschen Variablen und einer **while**-Schleife. Die Struktur ist folgendermaßen:

```
boolean fertig = false;  
while(!fertig) {  
    tue etwas  
    if(Ende-Bedingung) {  
        fertig = true;  
    }  
    else {  
        tue etwas anderes  
    }  
}
```

Dieses Programmiermuster ist eine Abwandlung der **while**-Schleife, die wir in Abschnitt 4.10 diskutiert haben. Wir benutzen **fertig** als einen Signalgeber, der

den Wert **true** bekommt, wenn wir die Schleife beenden wollen (und damit das ganze Programm). Zu Anfang stellen wir sicher, dass **fertig** den Wert **false** hat. (Erinnern Sie sich daran, dass ein Ausrufezeichen den *Nicht*-Operator darstellt!)

Der Hauptteil der Schleife – der Teil, der wiederholt ausgeführt wird, solange wir noch nicht fertig sind – besteht aus drei Anweisungen, wenn wir die Prüfung der Ende-Bedingung auslassen:

```
String eingabe = leser.gibEingabe();
...
String antwort = beantworter.generiereAntwort();
System.out.println(antwort);
```

In dieser Schleife wird somit wiederholt

- eine Benutzereingabe gelesen
- der Beantworter um eine Antwort gefragt
- diese Antwort ausgegeben

(Sie werden sicherlich bemerkt haben, dass die Antwort überhaupt nicht von der Eingabe abhängig ist! Das ist etwas, was wir ganz sicher ändern werden.)

Der letzte zu untersuchende Teil ist die Prüfung der Ende-Bedingung. Die Idee ist, dass das Programm enden soll, sobald der Benutzer „ade“ eingegeben hat. Die entsprechende Stelle im Quelltext lautet:

```
String eingabe = leser.gibEingabe();
if(eingabe.startsWith("ade")) {
    fertig = true;
}
```

Wenn Sie diese Abschnitte einzeln verstanden haben, dann sollten Sie noch einmal einen Blick auf die Methode **starten** in Listing 6.1 werfen, um zu sehen, ob Sie auch das Zusammenspiel verstehen.

In dem letzten Quelltextabschnitt, den wir oben betrachtet haben, wird eine Methode **startsWith** benutzt. Da diese Methode an der Variablen **eingabe** aufgerufen wird, die auf ein Objekt vom Typ **String** verweist, muss dies wohl eine Methode der Klasse **String** sein. Aber was tut diese Methode? Und wie finden wir das heraus?

Wir könnten einfach durch Deuten des Namens **startsWith** (englisch für „beginnt mit“) vermuten, dass sie prüft, ob die Zeichenkette mit dem Wort „ade“ beginnt. Wir können dies durch Probieren belegen. Starten Sie das Kundendienstsystem erneut und tippen Sie „ade, ade“ oder „adele“. Sie werden feststellen, dass beide Versionen das System beenden. Beachten Sie aber, dass eine Eingabe wie „Ade“ oder „ade“ – mit einem Großbuchstaben oder einem Leerzeichen am Anfang – nicht als beginnend mit „ade“ erkannt wird. Das könnte etwas lästig für den Benutzer sein, aber es wird sich herausstellen, dass wir dies recht leicht beheben können, sobald wir etwas mehr über die Klasse **String** wissen.

Wie bekommen wir mehr Informationen über die Methode **startsWith** oder andere Methoden der Klasse **String**?

## 6.3 Die Klassendokumentation lesen

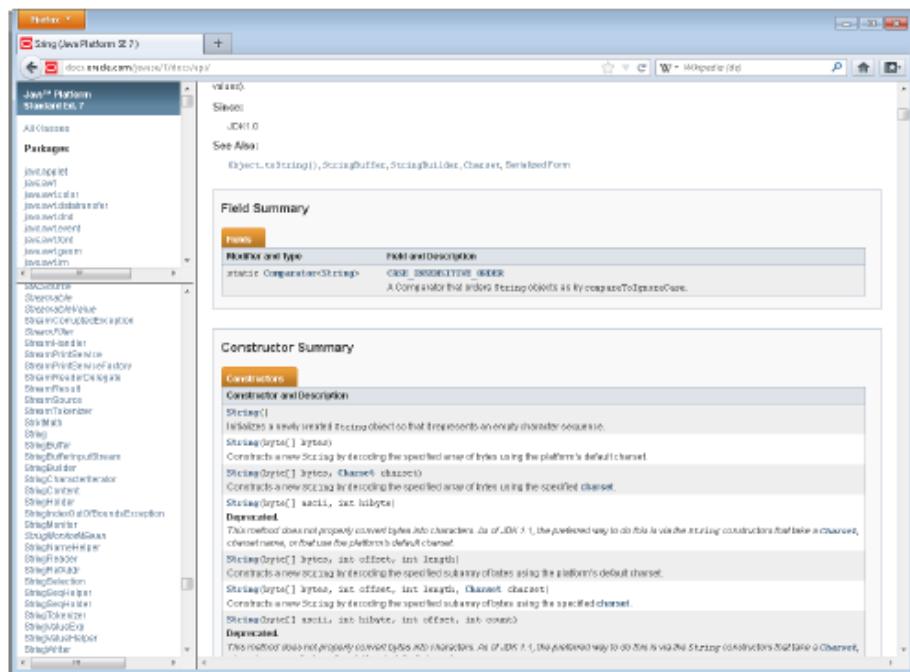
Die Klasse **String** ist eine Klasse aus der Java-Standardbibliothek. Wir können mehr über sie erfahren, indem wir die Bibliotheksdokumentation der Klasse **String** nachlesen.

Wählen Sie dazu den Eintrag JAVA KLASSENBIBLIOTHEKEN aus dem Menü HILFE in BlueJ. Dies öffnet einen Webbrowser, der die Hauptseite der Java-API-Dokumentation (API steht für Application Programmer Interface, die Schnittstelle für Anwendungsprogrammierer) anzeigt.<sup>1</sup>

Der Webbrowser zeigt drei Bereiche. In der linken oberen Ecke sehen Sie eine Liste der Pakete. Darunter sehen Sie eine Liste aller Klassen in der Java-Bibliothek. Der große Bereich auf der rechten Seite zeigt Detailinformationen über das selektierte Paket oder die selektierte Klasse.

Suchen Sie in der Liste auf der linken Seite nach der Klasse **String** und selektieren Sie sie. Der Bereich auf der rechten Seite zeigt daraufhin die Dokumentation der Klasse **String** an (Abbildung 6.3).

**Konzept**  
Die Dokumentation der Java-Standardbibliothek zeigt Informationen über alle Klassen in der Bibliothek. Die Benutzung dieser Dokumentation ist unerlässlich für einen vernünftigen Gebrauch der Bibliotheksklassen.



**Abbildung 6.3**  
Dokumentation der Java-Standardbibliothek.

<sup>1</sup> Per Voreinstellung greift diese Funktion auf die Dokumentation im Internet zu. Dies funktioniert nicht, wenn Ihr Computer keinen Netzzugang hat. BlueJ kann so konfiguriert werden, dass es auf eine lokale Kopie der Java-API-Dokumentation zugreift. Dies ist zu empfehlen, da es die Zugriffe beschleunigt und ohne einen Netzzugang funktioniert. Lesen Sie Anhang A für weitere Details.

**Übung 6.2** Untersuchen Sie die Dokumentation von `String`. Sehen Sie sich dann die Dokumentation einiger anderer Klassen an. Wie ist die Dokumentation strukturiert? Welche Abschnitte haben alle Klassenbeschreibungen? Welchem Zweck dienen sie?

**Übung 6.3** Suchen Sie die Methode `startsWith` in der Dokumentation von `String`. Es gibt sie in zwei Versionen. Beschreiben Sie mit eigenen Worten, was sie tun und welche Unterschiede zwischen ihnen bestehen.

**Übung 6.4** Gibt es eine Methode in der Klasse `String`, die testet, ob eine Zeichenkette mit einem gegebenen Suffix endet? Wenn ja, wie heißt sie und welche Parameter und welchen Ergebnistyp hat sie?

**Übung 6.5** Gibt es eine Methode in der `String`-Klasse, die die Anzahl der Zeichen in einer Zeichenkette liefert? Wenn ja, wie heißt sie und welche Parameter hat sie?

**Übung 6.6** Wenn Sie Methoden für die beiden vorigen Aufgaben gefunden haben, wie haben Sie dies getan? Ist es einfach oder schwierig, nach Methoden zu suchen? Warum?

### 6.3.1 Schnittstellen versus Implementierungen

#### Konzept

Die **Schnittstelle** einer Klasse beschreibt, was eine Klasse leistet und wie sie benutzt werden kann, ohne dass ihre Implementierung sichtbar wird.

Sie werden feststellen, dass die Dokumentation unterschiedliche Informationen liefert. Sie enthält, neben anderen Dingen:

- den Namen der Klasse
- eine allgemeine Beschreibung des Zwecks der Klasse
- eine Liste der Konstruktoren und Methoden der Klasse
- die Parameter und Ergebnistypen für jeden Konstruktor und jede Methode
- eine Beschreibung des Zwecks jedes Konstruktors und jeder Methode

Diese Informationen werden zusammengefasst als die *Schnittstelle* der Klasse bezeichnet. Beachten Sie, dass die Schnittstelle *nicht* den Quelltext zeigt, der die Klasse implementiert. Wenn eine Klasse gut beschrieben ist (genauer, wenn ihre Schnittstelle gut beschrieben ist), dann braucht ein Programmierer den Quelltext der Klasse nicht zu sehen, um die Klasse zu benutzen. Die Schnittstelle liefert alle notwendigen Informationen. Dies ist wieder einmal ein Fall von Abstraktion in Aktion.

#### Konzept

Der komplette Quelltext, der eine Klasse definiert, wird als die **Implementierung** dieser Klasse bezeichnet.

Der Quelltext hinter den Kulissen, der die Klasse ihren Dienst erfüllen lässt, wird als die *Implementierung* der Klasse bezeichnet. Normalerweise arbeitet ein Programmierer an der Implementierung von genau einer Klasse, während er einige andere Klassen über ihre Schnittstellen benutzt.

Die Unterscheidung zwischen der Schnittstelle und der Implementierung einer Klasse ist ein sehr wichtiges Konzept, das uns in diesem und den folgenden Kapiteln immer wieder begegnen wird.

**Hinweis**

Der Begriff *Schnittstelle* hat mehrere Bedeutungen im Bereich der Programmierung mit Java. Er wird benutzt, um den öffentlich sichtbaren Anteil einer Klasse zu beschreiben (so haben wir ihn gerade kennengelernt), hat aber auch noch andere Bedeutungen: Die Benutzungsschnittstelle (oft eine grafische Benutzungsschnittstelle) wird auch häufig einfach nur als *die Schnittstelle* bezeichnet. Weiterhin verfügt Java noch über ein Sprachkonstrukt namens Interface, das ins Deutsche übersetzt auch Schnittstelle heißt (dieses wird in Kapitel 12 besprochen). Wenn wir über das Java-Sprachkonstrukt reden, werden wir den englischen Begriff verwenden.

Es ist wichtig, die unterschiedlichen Bedeutungen des Begriffs *Schnittstelle* in den verschiedenen Zusammenhängen auseinanderhalten zu können.

Der Begriff der Schnittstelle ist auch auf einzelne Methoden anwendbar. Beispielsweise zeigt uns die Dokumentation der Klasse `String` die Schnittstelle der Methode `length`:

`public int length()`

*Returns the length of this string. The length is equal to the number of Unicode code units in the string.*

*Specified by:*

*length in interface CharSequence*

*Returns:*

*the length of the sequence of characters represented by this object.*

Die Schnittstelle einer Methode besteht aus dem Kopf der Methode und einem Kommentar (der hier kursiv angegeben ist). Zum Kopf einer Methode gehören (in dieser Reihenfolge):

- ein Zugriffsmodifikator (hier `public`) – wir werden diesen noch diskutieren
- der Ergebnistyp der Methode (hier `int`)
- der Name der Methode
- eine Liste von Parametern (die in diesem Beispiel leer ist); der Name und die Parameter werden zusammen auch als die *Signatur* der Methode bezeichnet.

Die Schnittstelle enthält all die Informationen, die wir für die Benutzung dieser Methode benötigen.

### 6.3.2 Methoden von Bibliotheksklassen benutzen

Zurück zu unserem Kundendienstsyste. Wir wollen nun die Verarbeitung der Eingabe etwas verbessern. Wir haben in der obigen Diskussion festgestellt, dass sich das System nicht sehr tolerant verhält. Wenn wir beispielsweise „Ade“ oder „ade“

statt „ade“ eintippen, wird das Wort nicht erkannt. Eine Sache, die wir beachten müssen, ist, dass ein **String**-Objekt nach seiner Erzeugung nicht mehr geändert werden kann. Wir wollen dies ändern, indem wir den vom Benutzer eingegebenen Text so anpassen, dass all diese Variationen als „ade“ erkannt werden.

Die Dokumentation der Klasse **String** gibt uns die Auskunft, dass sie über eine Methode **trim** verfügt, mit der Leerzeichen zu Beginn und am Ende einer Zeichenkette abgeschnitten werden können. Wir können diese Methode verwenden, um den zweiten Problemfall zu behandeln.

**Übung 6.7** Finden Sie die Methode **trim** in der Dokumentation der Klasse **String**. Schreiben Sie den Kopf dieser Methode auf. Schreiben Sie ein Beispiel für den Aufruf dieser Methode an einer **String**-Variablen **text** auf.

### Konzept

Ein Objekt wird als **unveränderlich** bezeichnet, wenn sein Inhalt oder Zustand nicht mehr geändert werden kann, nachdem es erzeugt wurde. **String** ist ein Beispiel für eine Klasse, die unveränderliche Objekte definiert.

Ein wichtiges Merkmal von **String**-Objekten ist, dass sie *unveränderlich* sind – das heißt, sie können nicht mehr geändert werden, nachdem sie erzeugt wurden. Beachten Sie vor allem, dass die Methode **trim** beispielsweise eine neue Zeichenkette zurückliefert und nicht die ursprüngliche Zeichenkette ändert. Deshalb sollten Sie den nachfolgenden Einschub mit den Fallstricken sorgfältig lesen.

### Fallstrick

Häufig wird in Java der Fehler gemacht, eine Zeichenkette verändern zu wollen – zum Beispiel durch folgende Anweisung

```
eingabe.toUpperCase();
```

Dies ist nicht korrekt (Zeichenketten können nicht geändert werden), erzeugt aber leider keinen Fehler. Die Anweisung hat einfach keine Auswirkung und die Eingabezeichenkette bleibt unverändert.

Die Methode **toUpperCase** sowie andere **String**-Methoden ändern nicht die ursprüngliche Zeichenkette, sondern liefern stattdessen eine *neue* Zeichenkette zurück, die gleich der alten ist einschließlich einiger Änderungen (hier mit den Zeichen, die in Großbuchstaben geändert wurden). Wenn wir wollen, dass unsere Variable **eingabe** geändert wird, dann müssen wir dieses neue Objekt wie folgt wieder der Variablen zuweisen (wobei wir uns der alten Variablen entledigen):

```
eingabe = eingabe.toUpperCase();
```

Das neue Objekt könnte auch einer anderen Variablen zugewiesen oder anderweitig verarbeitet werden.

Nachdem wir uns die Schnittstelle der Methode **trim** angesehen haben, wissen wir, dass wir die Leerzeichen aus der Eingabe mit folgendem Aufruf entfernen können:

```
eingabe = eingabe.trim();
```

Diese Anweisung veranlasst das **String**-Objekt, auf das **eingabe** verweist, eine neue Zeichenkette zu erzeugen, in der führende und anhängende Leerzeichen entfernt sind. Diese neue Zeichenkette wird dann in der Variablen **eingabe** gespeichert, da wir die alte nicht mehr benötigen. Somit verweist **eingabe** nach Ausführung dieser Zeile auf eine Zeichenkette, die an beiden Enden keine Leerzeichen mehr hat.

Wir können diese Zeile nun in unseren Quelltext einfügen, sodass er folgendermaßen aussieht:

```
String eingabe = leser.gibEingabe();
eingabe = eingabe.trim();

if(eingabe.startsWith("ade")) {
    fertig = true;
}
else {
    ... Quelltext hier ausgelassen
}
```

Die ersten beiden Zeilen können auch zu einer zusammengefasst werden:

```
String eingabe = leser.gibEingabe().trim();
```

Der Effekt dieser Zeile ist identisch mit den ersten beiden Zeilen oben. Die rechte Seite sollte gelesen werden, als wäre sie folgendermaßen geklammert:

```
(leser.gibEingabe()).trim();
```

Welche Version Sie bevorzugen, ist überwiegend eine Frage des Geschmacks. Die Entscheidung sollte vor allem in Hinsicht auf Lesbarkeit getroffen werden: Benutzen Sie die Version, die Sie einfacher zu lesen und zu verstehen finden. Programmieranfänger bevorzugen häufig die zweizeilige Version, während erfahrene Programmierer sich an die einzeilige Version gewöhnt haben.

**Übung 6.8** Implementieren Sie diese Verbesserung in Ihrer Version des Projekts **Technischer-Kundendienst1**. Testen Sie, ob anschließend Leerzeichen vor oder nach „ade“ toleriert werden.

Wir haben nun zwar das Problem mit den Leerzeichen um die Eingabe gelöst, aber das Problem mit den Großbuchstaben besteht weiterhin. Eine weitere Untersuchung der Dokumentation der Klasse **String** führt jedoch zu einer möglichen Lösung, denn dort ist eine Methode **toLowerCase** beschrieben.

**Übung 6.9** Verbessern Sie die Implementierung der Klasse **Kundendienstsystem** im Projekt **Technischer-Kundendienst1** so, dass die Groß-/Kleinschreibung bei der Eingabe ignoriert wird. Benutzen Sie dazu die Methode **toLowerCase** der Klasse **String**. Beachten Sie auch hier wieder, dass diese Methode nicht die Zeichenkette ändert, an der sie aufgerufen wird, sondern dass der Aufruf zur Erzeugung einer neuen mit leicht geändertem Inhalt führt.

### 6.3.3 Zeichenketten auf Gleichheit prüfen

Eine alternative Lösung wäre gewesen, zu prüfen, ob die eingegebene Zeichenkette die Zeichenkette „ade“ tatsächlich *ist*, und nicht, ob sie lediglich mit ade beginnt. Ein (falscher!) Versuch, dies zu formulieren, könnte so aussehen:

```
if (eingabe == "ade") { // funktioniert nicht immer!
    ...
}
```

Das Problem ist, dass es mehrere **String**-Objekte geben kann, die alle denselben Text repräsentieren. Zwei **String**-Objekte könnten beispielsweise beide aus den Zeichen „ade“ bestehen. Der Gleichheits-Operator (`=`) überprüft, ob seine beiden Operanden sich *auf dasselbe Objekt* beziehen, nicht, ob sie den gleichen Wert haben! Dies ist ein wichtiger Unterschied.

In unserem Beispiel wollen wir nicht wissen, ob die Variable **eingabe** und die Zeichenkettenkonstante „ade“ dasselbe Objekt bezeichnen, sondern, ob sie den gleichen Wert haben. Die Benutzung des Operators `=` ist somit falsch. Sie könnte **false** liefern, auch wenn der Wert der Variablen **eingabe** „ade“ ist.<sup>2</sup>

Die Lösung liegt in der Verwendung der Methode **equals**, die in der Klasse **String** definiert ist. Diese Methode überprüft, ob zwei **String**-Objekte den gleichen Inhalt haben. Der korrekte Quelltext sieht also so aus:

```
if(eingabe.equals("ade")) {
    ...
}
```

Dies kann natürlich auch mit den Methoden **trim** und **toLowerCase** kombiniert werden.

#### Fallstrick

Das Vergleichen von **String**-Objekten mit dem Operator `=` kann zu unerwarteten Ergebnissen führen. Als Grundregel gilt, dass Zeichenketten praktisch immer mit **equals** statt mit `==` verglichen werden sollten.

**Übung 6.10** Finden Sie die Methode **equals** in der Dokumentation der Klasse **String**. Welchen Ergebnistyp hat diese Methode?

**Übung 6.11** Ändern Sie Ihre Implementierung so, dass sie die Methode **equals** anstelle von **startsWith** verwendet.

---

<sup>2</sup> Leider führt aufgrund von Javas **String**-Implementierung der Vergleich zweier verschiedener **String**-Objekte gleichen Inhalts mit `==` oft irreführenderweise zur „korrekten“ Antwort. Trotzdem sollten Sie zum Vergleichen der Inhalte zweier **String**-Objekte niemals `=` verwenden.

## 6.4 Zufälliges Verhalten einbringen

Bisher haben wir einige kleine Verbesserungen am Projekt *Technischer-Kundendienst* vorgenommen, aber insgesamt ist das System noch sehr simpel. Eines der Hauptprobleme ist, dass es immer dieselbe Antwort liefert, unabhängig von der Eingabe durch den Benutzer. Wir werden dies nun verbessern, indem wir einige plausible Antworten definieren. Aus diesen Antworten lassen wir zufällig eine auswählen, wenn eine Anfrage gestellt wird. Dazu müssen wir die Klasse **Beantworter** in unserem Projekt erweitern.

Wir gehen folgendermaßen vor: Wir legen einige Antworten in einer **ArrayList** ab, generieren eine Zufallszahl und benutzen diese als Index in der Antwortliste, um eine der Antworten auszuwählen. In dieser Version werden wir die Antwort noch nicht von der Eingabe des Benutzers abhängig machen (das kommt später), aber zumindest sind die Antworten dann etwas abwechslungsreicher und sehen deutlich besser aus.

Dazu müssen wir herausfinden, wie man Zufallszahlen generieren kann.

### Zufall und Pseudozufall

Die Erzeugung von Zufallszahlen auf einem Computer ist nicht so einfach, wie man es vermuten könnte. Da Computer nach wohldefinierten deterministischen Verfahren arbeiten, die darauf beruhen, dass alle Berechnungen vorhersagbar und wiederholbar sind, gibt es eigentlich keinen Platz für zufälliges Verhalten in Computern.

Wissenschaftler haben im Laufe der Zeit viele Algorithmen vorgeschlagen, mit denen zufällig wirkende Folgen von Zahlen erzeugt werden können. Diese Zahlen sind tatsächlich nicht zufällig, sondern folgen lediglich sehr komplizierten Regeln. Deshalb werden sie auch Pseudozufallszahlen genannt.

In einer Sprache wie Java wurde die Erzeugung von Pseudozufallszahlen glücklicherweise durch eine Bibliotheksklasse implementiert, sodass wir, um Zufallszahlen zu bekommen, lediglich diese Klasse benutzen müssen.

Wenn Sie mehr zu diesem Thema lesen wollen, dann suchen Sie im Web nach den Schlüsselwörtern „Zufallszahlen Computer“ oder auf Englisch „pseudo random numbers“.

### 6.4.1 Die Klasse Random

Die Klassenbibliothek von Java enthält eine Klasse **Random** (englisch für „Zufall“), die für unser Projekt nützlich ist.

**Übung 6.12** Finden Sie die Klasse **Random** in der Dokumentation der Java-Bibliothek. In welchem Paket ist sie zu finden? Was tut sie? Wie erzeugen Sie eine Instanz? Wie generieren Sie eine Zufallszahl? Möglicherweise verstehen Sie nicht alle Erläuterungen in der Dokumentation. Versuchen Sie nur die Dinge herauszufinden, die Sie unmittelbar benötigen.

**Übung 6.13** Versuchen Sie, einen Quelltextabschnitt auf ein Blatt Papier zu schreiben, der Ihnen mithilfe dieser Klasse eine Zufallszahl generiert.

Um eine Zufallszahl zu erzeugen, müssen wir

- eine Instanz der Klasse `Random` erzeugen
- an dieser Instanz eine Methode aufrufen, um eine Zufallszahl zu erhalten

Ein Blick auf die Dokumentation sagt uns, dass es mehrere Methoden `next-`*Irgendetwas* gibt, die Zufallszahlen verschiedener Typen erzeugen. Diejenige zur Erzeugung von ganzen Zufallszahlen heißt `nextInt`.

Der folgende Quelltextabschnitt zeigt, wie eine ganze Zufallszahl generiert und ausgegeben werden kann:

```
Random zufallsgenerator;  
zufallsgenerator = new Random();  
int index = zufallsgenerator.nextInt();  
System.out.println(index);
```

In diesem Abschnitt wird eine neue Instanz der Klasse `Random` erzeugt und in der Variablen `zufallsgenerator` abgelegt. Anschließend wird an diesem Objekt die Methode `nextInt` aufgerufen, um eine Zufallszahl zu erhalten; diese wird in der Variablen `index` abgelegt und schließlich ausgegeben.

**Übung 6.14** Schreiben Sie Quelltext (in BlueJ), der die Erzeugung von Zufallszahlen testet. Definieren Sie dazu eine Klasse `ZufallszahlenTester`. Sie können diese Klasse im Projekt `Technischer-Kundendienst` anlegen oder ein eigenes Projekt dafür anlegen. Implementieren Sie in der Klasse zwei Methoden: `eine-ZufallszahlAusgeben` (die eine Zufallszahl ausgeben soll) und `zufallszahlen-Ausgeben(int anzah)` (die einen Parameter bekommt, der angibt, wie viele Zufallszahlen generiert und ausgegeben werden sollen).

Beachten Sie, dass Ihre Klasse nur eine Instanz der Klasse `Random` erzeugen und diese in einem Datenfeld ablegen sollte. Erzeugen Sie nicht für jede Zufallszahl eine neue Instanz von `Random`.

#### 6.4.2 Zufallszahlen mit eingeschränktem Wertebereich

Die Zufallszahlen, die wir bisher gesehen haben, liegen im Wertebereich der ganzen Zahlen von Java (von `-2147483648` bis `2147483647`). Das ist zum Experimentieren ganz okay, aber nicht sehr nützlich. Sehr häufig benötigen wir Zufallszahlen aus einem definierten Wertebereich.

Die Klasse `Random` bietet eine Methode an, die dies unterstützt. Sie heißt ebenfalls `nextInt`, hat jedoch einen Parameter, mit dem der gewünschte Wertebereich angegeben werden kann.

**Übung 6.15** Finden Sie die Methode `nextInt` in der Klasse `Random`, mit der Sie den Wertebereich der Zufallszahlen angeben können. Aus welchem Wertebereich stammen die erzeugten Zahlen, wenn Sie die Methode mit dem Parameter `100` aufrufen?

**Übung 6.16** Schreiben Sie eine Methode `wuerfeln` in Ihrer Klasse `ZufallszahlenTester`, die Werte von `1` bis einschließlich `6` zurückliefert.

**Übung 6.17** Schreiben Sie eine Methode `gibAntwort`, die zufällig eine der Zeichenketten "`ja`", "`nein`" und "`vielleicht`" zurückliefert.

**Übung 6.18** Erweitern Sie Ihre Methode `gibAntwort` so, dass sie eine `ArrayList` mit einer beliebigen Anzahl an Antworten benutzt, aus denen sie zufällig eine auswählt.

Wenn Sie eine Methode zur Erzeugung von Zufallszahlen aus einem bestimmten Wertebereich benutzen, dann sollten Sie sehr genau darauf achten, ob die Grenzen des Wertebereichs als *inklusiv* oder *exklusiv* angesehen werden. Die Methode `nextInt(int n)` in der Klasse `Random` beispielsweise spezifiziert, dass sie Zahlen aus dem Bereich von `0` (*inklusiv*) bis `n` (*exklusiv*) erzeugt. Der Wert `0` ist somit ein mögliches Ergebnis eines Aufrufs, während der angegebene Wert für `n` niemals zurückgeliefert wird. Die höchste gelieferte Zahl ist `n-1`.

**Übung 6.19** Fügen Sie Ihrer Klasse `ZufallszahlenTester` eine Methode hinzu, die einen Parameter `max` erhält und eine Zufallszahl aus dem Bereich `1` bis `max` liefert (*inklusive*).

**Übung 6.20** Fügen Sie Ihrer Klasse `ZufallszahlenTester` eine Methode hinzu, die zwei Parameter `min` und `max` erhält und eine Zufallszahl aus dem Bereich von `min` bis `max` liefert (*inklusive*). Schreiben Sie Ihre Methode aus der vorigen Übung so um, dass sie nun diese neue Methode benutzt, um ihr Ergebnis zu erzeugen. Beachten Sie, dass diese Methode nicht notwendigerweise mithilfe einer Schleife implementiert werden muss.

**Übung 6.21** Sehen Sie sich die Details der Klasse `SecureRandom` an, die im `java.security`-Paket definiert ist. Könnte diese Klasse anstelle der `Random`-Klasse verwendet werden? Warum sind Zufallszahlen für kryptografische Sicherheit wichtig?

### 6.4.3 Zufällige Antworten generieren

Wir können nun zur Klasse `Beantworter` zurückkehren, um diese zufällig eine Antwort aus einer Menge von vorgegebenen Antworten auswählen zu lassen. Listing 6.2 zeigt den Quelltext der Klasse `Beantworter` in unserer ersten Version.

Wir werden diesen Quelltext nun um einige Zeilen anreichern, die

- ein Datenfeld vom Typ `Random` für einen Zufallszahlengenerator deklarieren
- ein Datenfeld vom Typ `ArrayList` für die möglichen Antworten deklarieren

- im Konstruktor von **Beantworter** ein **Random**- und ein **ArrayList**-Objekt erzeugen
- die Antwortliste mit einigen Antworten füllen
- eine Antwort zufällig auswählen und zurückliefern, wenn **generiereAntwort** aufgerufen wird

Listing 6.3 zeigt eine Version der Klasse **Beantworter** mit diesen Ergänzungen.

### Listing 6.3

Der Quelltext der Klasse **Beantworter** mit zufälligen Antworten.

```
import java.util.ArrayList;
import java.util.Random;

/**
 * Die Klasse Beantworter beschreibt Exemplare, die
 * automatische Antworten generieren. Dies ist die zweite Version
 * dieser Klasse. In dieser Version generieren wir Antworten,
 * indem wir zufällig eine Antwortphrase aus einer Liste auswählen.
 *
 * @author Michael Kolling und David J. Barnes
 * @version 0.2 (2016.02.29)
 */
public class Beantworter
{
    private Random zufallsgenerator;
    private ArrayList<String> antworten;

    /**
     * Erzeuge einen Beantworter
     */
    public Beantworter()
    {
        zufallsgenerator = new Random();
        antworten = new ArrayList<>();
        antwortlisteFuellen();
    }

    /**
     * Generiere eine Antwort.
     *
     * @return einen String, der die Antwort enthält
     */
    public String generiereAntwort()
    {
        // Erzeuge eine Zufallszahl, die als Index in der Liste der
        // Antworten benutzt werden kann. Die Zahl wird im Bereich von
        // null (inklusiv) bis zur Größe der Liste (exklusiv) liegen.
        int index = zufallsgenerator.nextInt(antworten.size());
        return antworten.get(index);
    }

    /**
     * Generiere eine Liste von Standardantworten, aus denen wir eine
     * auswählen können, wenn wir keine bessere Antwort wissen.
     */
    private void antwortlisteFuellen()
    {
        antworten.add("Das klingt seltsam. Können Sie das ausführlicher beschreiben?");
        antworten.add("Bisher hat sich noch kein Kunde darüber beschwert.\n" +
                    "Welche Systemkonfiguration haben Sie?");
        antworten.add("Ich brauche ich etwas ausführlichere Angaben.");
        antworten.add("Haben Sie geprüft, ob Sie einen Konflikt mit einer DLL haben?");
        antworten.add("Das steht im Handbuch. Haben Sie das Handbuch gelesen?");
        antworten.add("Das klingt alles etwas Wischi-Waschi. Haben Sie einen Experten\n" +
                    "in der Nähe, den das besser beschreiben kann?");
        antworten.add("Das ist kein Fehler, das ist eine Systemeigenschaft!");
        antworten.add("Könnten Sie es anders erklären?");
        antworten.add("Haben Sie versucht, die App auf Ihrem Handy auszuführen?");
        antworten.add("Ich habe gerade StackOverflow überprüft - das ist auch keine Lösung.");
    }
}
```

In dieser Version haben wir den Teil, der die Antwortliste befüllt, in eine eigene Methode **antwortlisteFuellen** ausgelagert, die im Konstruktor aufgerufen wird. Dies garantiert, dass die Antwortliste befüllt wird, sobald ein **Beantworter**-Objekt erzeugt wird, doch durch diese Trennung wird der Quelltext der Klasse lesbarer und übersichtlicher.

Der interessanteste Abschnitt in dieser Klasse ist die Methode `generiereAntwort`. Ohne die Kommentare sieht sie folgendermaßen aus:

```
public String generiereAntwort()
{
    int index = zufallsgenerator.nextInt(antworten.size());
    return antworten.get(index);
}
```

Die erste Zeile in dieser Methode erledigt dreierlei:

- Sie lässt sich über den Aufruf von `size` die Größe der Antwortliste geben.
- Sie generiert eine Zufallszahl zwischen 0 (inklusiv) und der Größe (exklusiv).
- Sie speichert die generierte Zufallszahl in der lokalen Variablen `index`.

Wenn Ihnen dies zu viel in einer Zeile ist, können Sie stattdessen auch schreiben:

```
int listengroesse = antworten.size();
int index = zufallsgenerator.nextInt(listengroesse);
```

Diese Anweisungen sind äquivalent zur ersten Zeile oben. Welche Version Sie bevorzugen, hängt davon ab, welche Sie besser lesbar finden.

Beachten Sie, dass dieser Abschnitt Zufallszahlen aus dem Bereich von `0` bis `listengroesse-1` (inklusiv) erzeugt. Dies passt wunderbar zu den gültigen Indizes für eine `ArrayList`. Sie erinnern sich, dass der Wertebereich für Indizes einer `ArrayList` mit der Größe `listengroesse` von `0` bis `listengroesse-1` reicht. Die erzeugte Zufallszahl liefert uns somit einen perfekt geeigneten Index, um zufällig auf einen Eintrag in der Liste zugreifen zu können.

Die letzte Zeile der Methode lautet:

```
return antworten.get(index);
```

In dieser Zeile geschehen zwei Dinge:

- Sie fragt die Antwort an Position `index` mit der `get`-Methode ab.
- Sie liefert die gewählte Zeichenkette als Ergebnis der Methode mit der `return`-Anweisung zurück.

Wenn Sie nicht vorsichtig sind, kann es leicht passieren, dass Ihr Programm Zufallszahlen generiert, die außerhalb des gültigen Indexbereichs einer `ArrayList` liegen. Wenn Sie mit einem solchen ungültigen Index versuchen, auf ein Element zuzugreifen, erhalten Sie eine `IndexOutOfBoundsException`.

#### 6.4.4 Die Dokumentation generischer Klassen

Bisher haben wir Sie aufgefordert, sich die Dokumentation der Klasse `String` aus dem Paket `java.lang` sowie die der Klasse `Random` aus dem Paket `java.util` anzusehen. Sie haben dabei möglicherweise bemerkt, dass einige Klassennamen in der Dokumentation etwas anders aussehen, wie beispielsweise `ArrayList<E>` oder `HashMap<K,V>`. Hinter den Klassennamen stehen hier Zusatzinformationen in spitzen Klammern. Solche Klassen heißen *parametisierte Klassen* oder *generische Klassen*. Die Angaben in den spitzen Klammern sagen uns, welche zusätzlichen Typnamen wir zur Vervollständigung angeben müssen, um diese Klassen zu benutzen.

Wir haben dies bereits in Kapitel 4 angewendet, als wir die Klasse `ArrayList` benutztten, indem wir sie mit Typnamen wie `String` parametrisiert haben:

```
private ArrayList<String> notizen;  
private ArrayList<Posten> studenten;
```

Weil wir eine `ArrayList` mit einem beliebigen Klassentyp parametrisieren können, ist dies entsprechend in der Dokumentation der Java API angegeben. Wenn Sie sich beispielsweise die Methodenliste der Klasse `ArrayList<E>` ansehen, entdecken Sie Methoden wie:

```
boolean add(E o)  
E get(int index)
```

Hier sehen wir, dass der Typ der Objekte, die wir in eine `ArrayList` mit `add` einfügen dürfen, von dem Typ abhängt, mit dem die Klasse parametrisiert wurde. Das Gleiche gilt für den Ergebnistyp der Methode `get`. Im Endeffekt sagt die Dokumentation, dass wir, wenn wir ein Objekt des Typs `ArrayList<String>` erzeugen, ein Objekt mit den folgenden beiden Methoden erhalten:

```
boolean add(String o)  
String get(int index)
```

Wenn wir hingegen ein Objekt vom Typ `ArrayList<Student>` erzeugen, dann hat es diese beiden Methoden:

```
boolean add(Student o)  
Posten get(int index)
```

Wir werden Sie später in diesem Kapitel noch auffordern, sich die Dokumentation weiterer parametrisierter Typen anzusehen.

## 6.5 Pakete und Importe

Es gibt noch zwei Zeilen zu Anfang der Quelltextdatei, die wir hier ansprechen müssen:

```
import java.util.ArrayList;  
import java.util.Random;
```

Wir haben die `import`-Anweisung zum ersten Mal in Kapitel 4 gesehen. Wir werden sie nun etwas genauer untersuchen.

Die Java-Klassen in einer Klassenbibliothek stehen nicht automatisch zur Verfügung, so wie die übrigen Klassen in unserem aktuellen Projekt. Stattdessen müssen wir in unserem Quelltext deklarieren, dass wir eine Klasse aus der Bibliothek benutzen möchten. Dies wird *Importieren* der Klasse genannt und geschieht entsprechend mit einer `import`-Anweisung. Die `import`-Anweisung hat die Form

```
import qualifizierter-Klassenname;
```

Da die Java-Bibliothek aus mehreren Tausend Klassen besteht, ist ein Strukturierungsmechanismus notwendig, mit dem der Umgang mit dieser Menge erleichtert wird. Java benutzt Pakete (*package*), um Bibliotheksklassen in zusammen-

gehörigen Gruppen zu bündeln. Pakete können geschachtelt werden, d.h., ein Paket kann weitere Pakete enthalten.

Die Klassen `ArrayList` und `Random` gehören beide zum Paket `java.util`. Diese Information steht in der Klassendokumentation. Der vollständige oder qualifizierte Name einer Klasse besteht aus dem Namen seines Pakets, gefolgt von einem Punkt und dem Klassennamen. Die qualifizierten Namen der beiden genannten Klassen sind somit `java.util.ArrayList` und `java.util.Random`.

Java erlaubt auch, dass ganze Pakete mit Anweisungen der Form

```
import Paketname.*;
```

importiert werden. Die folgende Anweisung würde somit alle Klassennamen aus dem Paket `java.util` importieren:

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

Die Klassen einzeln aufzuführen, so wie in unserer ersten Version, bedeutet etwas mehr Schreibarbeit, dokumentiert aber auch deutlicher, welche Klassen tatsächlich benutzt werden. Deshalb werden wir überwiegend dem Stil des ersten Beispiels folgen, also jede Klasse einzeln importieren.

Es gibt eine Ausnahme für die Importregeln: Einige Klassen werden so oft benötigt, dass sie praktisch in jede Klasse importiert werden müssen. Diese Klassen sind im Paket `java.lang` definiert und dieses Paket wird automatisch in jede Klasse importiert. `String` ist ein Beispiel für eine Klasse aus `java.lang`.

**Übung 6.22** Implementieren Sie die hier diskutierte Lösung mit den zufälligen Antworten in Ihrer Version des Projekts *Technischer-Kundendienst*.

**Übung 6.23** Was passiert, wenn Sie weitere Antworten in die Antwortliste einfügen (oder einige entfernen)? Wird die zufällige Auswahl einer Antwort weiterhin funktionieren? Warum oder warum nicht?

Die Lösung für diese Aufgaben finden Sie auch im Begleitmaterial unter dem Namen *Technischer-Kundendienst2*. Wir empfehlen jedoch, dass Sie diese Lösung selbst als eine Erweiterung der Basisversion implementieren.

## 6.6 Benutzung von Map-Klassen für Abbildungen

Wir haben nun eine Lösung für unser Kundendienstsysteem, die zufällige Antworten generiert. Das ist besser als unsere erste Version, aber noch nicht wirklich überzeugend. Insbesondere werden die Antworten nach wie vor nicht von den Eingaben des Benutzers beeinflusst. Das wollen wir nun verbessern.

Die Idee ist, dass wir eine Menge von Wörtern definieren, die typischerweise in Anfragen auftauchen, und diese Wörter mit entsprechenden Antworten ver-

knüpfen. Wenn die Eingabe des Benutzers eines der uns bekannten Wörter enthält, dann können wir eine passende Antwort geben. Das ist immer noch eine sehr ungenaue Methode, da sie weder auf den Inhalt einer Anfrage eingeht noch den Kontext mit einbezieht. Sie kann dennoch überraschend effektiv sein und es ist ein guter nächster Schritt.

Für diesen Schritt werden wir eine **HashMap** verwenden. Sie finden die Dokumentation der Klasse **HashMap** in der Java-Bibliothek. **HashMap** ist eine Spezialisierung von **Map**, die ebenfalls dokumentiert ist. Sie werden sehen, dass Sie beide lesen müssen, um zu verstehen, was eine **HashMap** ist und wie sie funktioniert.

**Übung 6.24** Was ist eine **HashMap**? Welchen Zweck hat sie und wie benutzt man sie?

Beantworten Sie diese Fragen schriftlich. Benutzen Sie die Dokumentation von **Map** und **HashMap** in der Java-Bibliothek, um diese Fragen zu beantworten. Beachten Sie, dass es schwierig sein kann, alles zu verstehen, da die Dokumentation dieser Klassen nicht sehr gut ist. Wir werden die Details in diesem Kapitel noch besprechen, aber bevor Sie weiterlesen, sollten Sie versuchen, möglichst viel auf eigene Faust herauszufinden.

**Übung 6.25** **HashMap** ist eine parametisierte Klasse. Benennen Sie die Methoden, die von den Typen abhängen, mit denen die Klasse parametrisiert wird. Denken Sie, dass derselbe Typ für beide Typparameter benutzt werden kann?

### 6.6.1 Das Konzept einer Map

#### Konzept

Eine **Map** ist eine Sammlung, die Schlüssel-Wert-Paare als Einträge enthält. Ein Wert kann ausgelesen werden, indem ein Schlüssel angegeben wird.

Eine Map ist eine Sammlung von Schlüssel-Wert-Paaren, wobei sowohl Schlüssel als auch Werte Objekte sind. Wie auch eine **ArrayList** kann eine **Map** eine beliebige Anzahl von Einträgen haben. Ein Unterschied zwischen einer **ArrayList** und einer **Map** ist, dass bei einer **Map** ein Eintrag nicht ein einzelnes Objekt ist, sondern ein Paar von Objekten. Ein solches Paar besteht aus einem *Schlüssel*-Objekt und einem *Wert*-Objekt.

Bei einer Map verwenden wir nicht einen Index, um ein Element einer Sammlung zu referenzieren (wie wir es bei einer **ArrayList** tun), sondern wir nehmen das Schlüssel-Objekt, um das Wert-Objekt zu bekommen. Im Deutschen verwenden wir den Begriff *Abbildung* für dieses Konzept: Ein Schlüssel wird auf einen Wert abgebildet.

Ein Alltagsbeispiel für eine solche Abbildung ist eine Kontaktliste. Eine Kontaktliste enthält Einträge, die Paare aus einem Namen und einer Telefonnummer sind. Wir verwenden eine Kontaktliste, indem wir einen Namen nachschlagen und dann die zugeordnete Nummer verwenden. Wir benutzen keinen Index – also die Position eines Eintrags in der Kontaktliste –, um sie zu finden.

Eine **Map** kann so strukturiert sein, dass das Nachschlagen eines Wertes für einen Schlüssel einfach sein kann. Bei einer Kontaktliste beispielsweise ist dies durch

die alphabetische Ordnung gewährleistet. Da die Einträge alphabetisch nach ihren Schlüsseln sortiert sind, ist das Finden eines Schlüssels und damit des zugeordneten Werts einfach. Die umgekehrte Suche (einen Schlüssel für einen Wert suchen, also einen Namen zu einer Telefonnummer) ist mit einer solchen Abbildung nicht so einfach. Wie im Falle einer Kontaktliste wäre die umgekehrte Suche zwar möglich, aber im Vergleich extrem zeitaufwendig. Abbildungen sind somit ideal für eine einseitig gerichtete Suche, wenn wir den Suchschlüssel kennen und den zugeordneten Wert benötigen.

## 6.6.2 Die Benutzung einer **HashMap**

**HashMap** ist eine spezifische Implementierung einer **Map**. Die wichtigsten Methoden einer **HashMap** sind **put** und **get**.

Die Methode **put** fügt der Abbildung einen Eintrag hinzu und **get** liefert für einen gegebenen Schlüssel einen Wert. Der folgende Quelltextabschnitt erzeugt eine **HashMap** und fügt drei Einträge ein. Jeder Eintrag ist ein Schlüssel-Wert-Paar, bestehend aus einem Namen und einer Telefonnummer.

```
HashMap<String, String> kontakte = new HashMap<>();
kontakte.put("Günter Schmidt", "(0531) 392 4587");
kontakte.put("Harald Lorant", "(089) 7386 6632");
kontakte.put("Werner Jauch", "(030) 4532 7761");
```

Wie auch bei einer **ArrayList** müssen wir, wenn wir eine **HashMap**-Variable deklarieren und ein **HashMap**-Objekt erzeugen, den Typ der Objekte angeben, die in der **Map** enthalten sein sollen; zusätzlich müssen wir hier auch den Typ des Schlüssels angeben. Für unsere Kontaktliste verwenden wir **String** sowohl für den Schlüsseltyp als auch für den Elementtyp, aber die beiden Typen müssen im Allgemeinen nicht gleich sein.

Wie wir bereits in Abschnitt 4.4.2 gesehen haben, müssen wir bei der Erzeugung von Objekten generischer Klassen und deren Zuweisung an Variablen die generischen Typen (hier **<String, String>**) nur einmal auf der linken Seite der Zuweisung angeben und können den Diamant-Operator in der Objekterzeugung auf der rechten Seite verwenden. Die generischen Typen für die Objekterzeugung werden dann von der Variablen Deklaration kopiert.

Die folgenden Anweisungen suchen die Telefonnummer von Harald Lorant und geben sie aus.

```
String nummer = kontakte.get("Harald Lorant");
System.out.println(nummer);
```

Beachten Sie, dass hier der Schlüssel (der Name „Harald Lorant“) an die **get**-Methode übergeben wird, um den Wert (die Telefonnummer) zu bekommen.

Lesen Sie die Dokumentation von **get** und **put** der Klasse **HashMap** erneut, um festzustellen, ob diese Erklärungen Ihrem aktuellen Verständnis entsprechen.

**Übung 6.26** Wie überprüfen Sie, wie viele Einträge in einer Abbildung vorhanden sind?

**Übung 6.27** Legen Sie eine Klasse **MapTester** an (entweder in Ihrem aktuellen Projekt oder in einem eigenen). Verwenden Sie in diesem Projekt eine **HashMap**, um eine Kontaktliste wie oben dargestellt zu implementieren. (Vergessen Sie nicht, `java.util.HashMap` zu importieren.) Implementieren Sie zwei Methoden in dieser Klasse:

```
public void nummerEintragen(String name, String nummer)
```

und

```
public String nummerSuchen(String name)
```

Diese Methoden sollten die Methoden `put` und `get` der Klasse **HashMap** verwenden, um ihre Funktionalität zu implementieren.

**Übung 6.28** Was passiert, wenn Sie bei einer solchen Abbildung einen Eintrag mit einem Schlüssel vornehmen, der bereits eingetragen ist?

**Übung 6.29** Was passiert, wenn Sie bei einer solchen Abbildung zwei Einträge mit demselben Wert, aber mit verschiedenen Schlüsseln vornehmen?

**Übung 6.30** Wie prüfen Sie, ob ein gegebener Schlüssel bereits in einer Abbildung eingetragen ist? (Geben Sie einen Java-Quelltext an.)

**Übung 6.31** Was passiert, wenn Sie einen Wert suchen und der Schlüssel in der Abbildung nicht existiert?

**Übung 6.32** Wie geben Sie alle Schlüssel aus, die aktuell in einer Abbildung gespeichert sind?

### 6.6.3 Benutzung einer Abbildung für das Kundendienstsystem

Für unser Kundendienstsystem können wir eine solche Abbildung gut gebrauchen, indem wir bekannte Wörter als Schlüssel nehmen und zugeordnete Antworten als Werte. Listing 6.4 zeigt ein Beispiel, in dem eine **HashMap** namens `antwortMap` erzeugt wird und drei Einträge vorgenommen werden. Beispielsweise ist das Wort „langsam“ mit folgendem Text verknüpft:

*Ich vermute, dass das mit Ihrer Hardware zu tun hat. Ein Upgrade für Ihren Prozessor sollte diese Probleme lösen. Haben Sie ein Problem mit unserer Software?*

Wenn nun jemand eine Anfrage stellt, die das Wort „langsam“ enthält, können wir diese Antwort suchen und ausgeben. Beachten Sie, dass die Antwort-Zeichenkette im Quelltext über mehrere Zeilen geht, die einzelnen Teile aber mit + verknüpft sind und deshalb als ein einzelner Wert in die **HashMap** eingetragen werden.

```

private HashMap<String, String> antwortMap;
...
public Beantworter()
{
    antwortMap = new HashMap<>();
    antwortMapBefuellen();
}

/**
 * Trage alle bekannten Stichwörter mit ihren verknüpften
 * Antworten in die Map 'antwortMap' ein.
 */
private void antwortMapBefuellen()
{
    antwortMap.put("langsam",
        "Ich vermute, dass das mit Ihrer Hardware zu tun hat. Ein Upgrade\n" +
        "für Ihren Prozessor sollte diese Probleme lösen.\n" +
        "Haben Sie ein Problem mit unserer Software?");

    antwortMap.put("fehler",
        "Wissen Sie, jede Software hat Fehler. Aber unsere Entwickler\n" +
        "arbeiten sehr hart daran, diese Fehler zu beheben.\n" +
        "Können Sie das Problem ein wenig genauer beschreiben?");

    antwortMap.put("teuer",
        "Unsere Preise sind absolute Marktpreise. Haben Sie sich mal umgesehen\n" +
        "und wirklich unser Leistungsspektrum verglichen?");
}

```

**Listing 6.4**

Ausgewählte Stichwörter mit Antworten verknüpfen.

Ein erster Versuch, eine Methode zum Erzeugen der Antworten zu schreiben, könnte so aussehen wie die Methode **generiereAntworten** unten. Zur Vereinfachung nehmen wir hier vorläufig an, dass der Benutzer lediglich ein einzelnes Wort (etwa „langsam“) eingegeben hat.

```

public String generiereAntwort(String wort)
{
    String antwort = antwortMap.get(wort);
    if(antwort != null) {
        return antwort;
    }
    else {
        // Wenn wir hierher gelangen, wurde das Stichwort nicht erkannt.
        // In diesem Fall wählen wir eine unserer Standardantworten.
        return standardantwortAuswählen();
    }
}

```

In diesem Quelltextabschnitt schlagen wir ein Wort, das der Benutzer eingegeben hat, in der Map nach. Wenn wir dort einen Eintrag finden, liefern wir die zugeordnete Antwort. Wenn wir keinen Eintrag für das Wort finden, rufen wir die Methode **standardantwortAuswählen** auf. Diese Methode kann die Anweisungen aus unserer vorherigen Version von **generiereAntwort** enthalten, die zufällig eine der Standardantworten auswählt (wie in Listing 6.3 dargestellt). Die neue Verarbeitungslogik ist somit, dass wir eine angemessene Antwort liefern, wenn wir ein Stichwort erkennen, ansonsten wählen wir zufällig eine unserer Standardantworten aus.

**Übung 6.33** Implementieren Sie die hier diskutierten Veränderungen in Ihrer Version des Kundendienstsystems. Testen Sie es, um ein Gefühl für die Verbesserung zu bekommen.

Dieser Ansatz, Stichwörter mit Antworten zu verknüpfen, funktioniert nur, solange der Benutzer statt ganzer Sätze nur einzelne Wörter eingibt. Die letzte Verbesserung unserer Anwendung wird sein, dass ein Benutzer wieder ganze Sätze eingeben kann und wir dann eine passende Antwort auswählen, wenn wir eines der Wörter in der Anfrage erkennen.

Dies stellt uns vor das Problem, dass wir Stichwörter in dem Satz erkennen müssen, den der Benutzer eingibt. In der momentanen Version wird die Benutzereingabe durch den **Eingabeleser** als eine einzelne Zeichenkette geliefert. Wir werden dies nun so ändern, dass der **Eingabeleser** eine Menge von Wörtern liefert. Technisch wird das eine Menge von Zeichenketten sein, von denen jede ein Wort aus der Benutzereingabe repräsentiert.

Wenn wir das hinbekommen, dann können wir diese Wortmenge an den **Beantworter** weiterreichen, der dann jedes Wort in der Menge daraufhin überprüft, ob es bekannt ist und mit einer Antwort verknüpft ist.

Um dies in Java umzusetzen, benötigen wir zweierlei: Wir müssen wissen, wie wir eine Zeichenkette mit einem ganzen Satz in einzelne Wörter zerlegen können, und wir müssen mit Mengen umgehen können. Diese Punkte werden in den nächsten beiden Abschnitten behandelt.

## 6.7 Der Umgang mit Mengen

### Konzept

Eine **Menge** (**Set**) ist eine Sammlung, in der jedes Element nur maximal einmal enthalten ist. Die Elemente einer Menge haben keine spezifische Ordnung.

Die Java-Bibliothek bietet verschiedene Varianten von Mengen (**set**), die durch unterschiedliche Klassen implementiert sind. Wir werden im Folgenden ein **HashSet** benutzen.

**Übung 6.34** Was sind die Gemeinsamkeiten und die Unterschiede zwischen einem **HashSet** und einer **ArrayList**? Benutzen Sie die Beschreibungen von **Set**, **HashSet**, **List** und **ArrayList** in der Dokumentation der Bibliothek für Ihre Untersuchung, da ein **HashSet** ein spezieller Fall eines **Set** ist und eine **ArrayList** ein spezieller Fall einer **List**.

Die beiden Dienstleistungen, die wir von einer Menge erwarten, sind das Eintragen von Elementen und das spätere Abfragen dieser Elemente. Glücklicherweise sind uns diese Anforderungen nicht neu. Betrachten Sie folgenden Quelltextabschnitt:

```
import java.util.HashSet;
import java.util.Iterator;
...
HashSet<String> meineMenge = new HashSet<>();
meineMenge.add("eins");
meineMenge.add("zwei");
meineMenge.add("drei");
```

Vergleichen Sie diese Anweisungen mit denen, die für das Einfügen der Elemente in eine **ArrayList** notwendig sind. Es gibt praktisch keinen Unterschied, außer dass wir diesmal ein **HashSet** erzeugen statt einer **ArrayList**. Nun wollen wir das Iterieren über die Menge betrachten:

```
for (String eintrag : meineMenge) {
    etwas mit diesem Eintrag tun
}
```

Auch hier sehen wir, dass dies die gleichen Anweisungen sind, die wir in Kapitel 4 für das Iterieren über eine **ArrayList** benutzt haben.

Kurz gesagt: Der Umgang mit den Sammlungen in Java ist für die verschiedenen Sammlungstypen sehr ähnlich. Sobald Sie den Umgang mit einer verstanden haben, können Sie alle benutzen. Die tatsächlichen Unterschiede liegen im Verhalten der jeweiligen Sammlung. Eine Liste (**List**) beispielsweise hält ihre Elemente in der gewünschten Reihenfolge, bietet den Zugriff über einen Index und kann daselbe Element mehrfach enthalten. Eine Menge (**Set**) hingegen definiert für ihre Elemente keine Reihenfolge (die **for-each**-Schleife kann die Elemente in einer anderen Reihenfolge liefern als die, in der sie eingefügt wurden) und garantiert, dass jedes Element nur maximal einmal in der Menge vorkommt. Wenn ein Element ein zweites Mal eingefügt wird, hat dies schlicht keinen Effekt.

## 6.8 Zeichenketten zerlegen

Nachdem wir den Umgang mit Mengen betrachtet haben, können wir nun untersuchen, wie wir eine Eingabezeile in mehrere Wörter zerlegen und diese in einer Menge ablegen können. Die Lösung ist als eine Version der Methode **gibEingabe** der Klasse **Eingabeleser** realisiert (Listing 6.5).

```
/** 
 * Lies eine Zeile von der Konsole und liefere sie als eine
 * Menge von Wörtern zurück.
 *
 * @return eine Menge von Zeichenketten, von denen jede ein
 *         von Benutzer getipptes Wort repräsentiert
 */
public HashSet<String> gibEingabe()
{
    System.out.print("> "); // Eingabebereitschaft anzeigen
    String eingabezeile = scanner.nextLine().trim().toLowerCase();

    String[] wortArray = eingabezeile.split(" "); //Trennung bei Leerzeichen

    // Wörter aus dem Array in das HashSet einfügen
    HashSet<String> woerter = new HashSet<String>();
    for(String wort : wortArray) {
        woerter.add(wort);
    }

    return woerter;
}
```

**Listing 6.5**

Die Methode **gibEingabe** liefert eine Menge von Wörtern.

Hier verwenden wir zusätzlich zur Klasse **HashSet** die Methode **split** der Klasse **String**, die ebenfalls in der Java-Standardsbibliothek definiert ist.

Die Methode `split` kann einen String in getrennte Substrings aufteilen und diese in einem String-Array zurückliefern (Arrays werden im nächsten Kapitel noch ausführlich besprochen). Der Parameter der Methode `split` definiert, an welcher Art von Zeichen der Original-String zerlegt werden soll. Wir haben definiert, dass wir unseren String an jedem Leerzeichen zerlegen wollen.

Die nächsten Zeilen erzeugen ein `HashSet` und kopieren die Wörter aus dem Array in dieses hinein, bevor das Set zurückgeliefert wird.<sup>3</sup>

**Übung 6.35** Die Methode `split` ist mächtiger, als sie in diesem Beispiel auf den ersten Blick aussieht. Wie können Sie exakt festlegen, wie ein String zerlegt werden soll? Geben Sie einige Beispiele an.

**Übung 6.36** Wie würden Sie die Methode `split` aufrufen, wenn Sie einen String an Leerzeichen oder Tabulatorzeichen zerlegen wollen? Wie können Sie einen String zerlegen, in dem die Wörter durch Doppelpunkte (:) getrennt sind?

**Übung 6.37** Worin unterscheidet sich das Ergebnis, wenn die Wörter statt in einem `HashSet` in einer `ArrayList` zurückgeliefert werden?

**Übung 6.38** Was passiert, wenn mehr als ein Leerzeichen zwischen zwei Wörtern steht (beispielsweise zwei oder drei Leerzeichen)? Gibt es ein Problem?

**Übung 6.39 Zusatzaufgabe.** Lesen Sie die Fußnote über die Methode `Arrays.asList`. Finden und lesen Sie die späteren Abschnitte über Klassenvariablen und Klassenmethoden. Erklären Sie in Ihren eigenen Worten, wie diese Lösung funktioniert.

**Übung 6.40** Welche weiteren Methoden bietet die Klasse `Arrays`?

**Übung 6.41** Erzeugen Sie eine Klasse namens `SortierenTest`. Erzeugen Sie in dieser Klasse eine Methode, die ein Array von `int`-Werten als Parameter entgegennimmt und die Elemente sortiert (kleinstes Element zuerst) auf der Konsole ausgibt.

## 6.9 Abschluss des Kundendienstsystems

Um alles zusammensetzen zu können, müssen wir nun auch noch die Klassen `Kundendienstsystem` und `Beantworter` so anpassen, dass sie mit einer Menge von Wörtern statt mit einer einzelnen Zeichenkette umgehen können. Listing 6.6 zeigt

---

3 Es gibt einen kürzeren und noch eleganteren Weg. Sie können schreiben:  
`HashSet<String> woerter = new HashSet<>(Arrays.asList(wortArray));`  
um alle vier Zeilen Quelltext zu ersetzen. Diese Lösung benutzt die Klasse `Arrays` aus der Standardbibliothek und eine *statische Methode* (häufig auch *Klassenmethode* genannt), die wir hier noch nicht diskutieren wollen. Wenn Sie neugierig sind, dann lesen Sie den Abschnitt 6.16 über Klassenmethoden und probieren diese Version aus.

die neue Version der Methode `starten` aus der Klasse `Kundendienstsystem`. Sie hat sich nicht sehr stark verändert. Die Änderungen sind im Einzelnen:

- Die Variable `eingabe` für das Ergebnis des Aufrufs `leser.gibEingabe()` ist nun vom Typ `HashSet`.
- Die Prüfung auf Beendigung der Anwendung wird mit der Methode `contains` der Klasse `HashSet` vorgenommen statt mit einer `String`-Methode. (Lesen Sie diese Methode in der Dokumentation nach.)
- Die Klasse `HashSet` muss mit einer `import`-Anweisung importiert werden (im Listing nicht gezeigt).

```
public void starten()
{
    boolean fertig = false;
    willkommenstextAusgeben();

    while(!fertig) {
        HashSet<String> eingabe = leser.gibEingabe();

        if(eingabe.contains("ade")) {
            fertig = true;
        } else {
            String antwort = beantworter.generiereAntwort(eingabe);
            System.out.println(antwort);
        }
    }
    abschiedstextAusgeben();
}
```

#### Listing 6.6

Die endgültige Version der Methode `starten`.

Schließlich müssen wir noch die Methode `generiereAntwort` in der Klasse `Beantworter` so erweitern, dass sie eine Menge von Wörtern als Parameter akzeptiert. Sie muss dann über diese Wörter iterieren und bei jedem überprüfen, ob es in der Map mit den bekannten Stichwörtern enthalten ist. Wenn eines der Wörter erkannt wird, liefern wir sofort die entsprechende Antwort zurück. Wenn wir keines der Wörter erkennen, liefern wir wie vorher eine der Standardantworten. Listing 6.7 zeigt diese Lösung.

```
public String generiereAntwort(HashSet<String> woerter)
{
    for (String wort : woerter) {
        String antwort = antwortMap.get(wort);
        if(antwort != null) {
            return antwort;
        }
    }
    // Wenn wir hierher gelangen, wurde keines der Eingabewörter erkannt.
    // In diesem Fall wählen wir eine unserer Standardantworten (die
    // wir geben, wenn uns nichts Besseres mehr einfällt).

    return standardantwortAuswählen();
}
```

#### Listing 6.7

Die endgültige Version der Methode `generiereAntwort`.

Dies ist die letzte Änderung an dieser Anwendung, die wir in diesem Kapitel diskutieren. Die Lösung im Projekt *Technischer-Kundendienst-komplett* enthält all diese Änderungen. Sie enthält auch eine größere Anzahl Verknüpfungen von Stichwörtern zu Antworten, als in diesem Kapitel gezeigt wird.

Es sind noch viele Verbesserungen dieser Anwendung denkbar. Wir werden diese hier nicht diskutieren. Stattdessen geben wir Verbesserungsvorschläge in den Übungen, die der Leser vornehmen kann. Einige davon sind recht anspruchsvolle Programmieraufgaben.

**Übung 6.42** Implementieren Sie die oben diskutierten Änderungen in Ihrer Version des Projekts.

**Übung 6.43** Fügen Sie Ihrer Anwendung weitere Wort-Antwort-Paare hinzu. Einige können Sie aus der gegebenen Lösung kopieren und einige selbst hinzufügen.

**Übung 6.44** Stellen Sie sicher, dass die gleiche Standardantwort nicht zweimal hintereinander ausgewählt wird.

**Übung 6.45** Einige Wörter werden auf dieselbe Antwort abgebildet. Erweitern Sie Ihr System so, dass es mit Synonymen und abgeleiteten Wörtern explizit umgeht, damit Sie nicht mehrere Einträge in der Map für dieselbe Antwort vornehmen müssen.

**Übung 6.46** Identifizieren Sie auch mehrere passende Wörter in der Benutzereingabe und reagieren Sie mit einer passenderen Antwort.

**Übung 6.47** Wenn kein Wort erkannt wird, können Sie auch auf andere Weise auf Wörter aus der Eingabe reagieren, beispielsweise mit Standardantworten auf Fragen mit „Wie“, „Warum“ und „Wer“.

## 6.10 Autoboxing und Wrapper-Klassen

Wir haben gesehen, dass die Sammlungsklassen (wie `ArrayList`) mit der geeigneten Parametrisierung Objekte jedes Objekttyps speichern können. Es bleibt jedoch ein Problem: Java hat auch einige Typen, die keine Objekttypen sind.

Wir erinnern uns, dass die primitiven Typen wie `int`, `boolean`, `char` etc. sich grundsätzlich von Objekttypen unterscheiden. Ihre Werte sind nicht Instanzen von Klassen und können normalerweise nicht in Sammlungen eingetragen werden.

Das ist schade. Es sind Situationen denkbar, in denen wir beispielsweise Listen von `int`-Werten oder Mengen von Zeichen (`char`) benötigen. Was können wir tun?

Die Lösung von Java für dieses Problem sind die sogenannten *Wrapper-Klassen* (vom Englischen *to wrap*: einpacken, umwickeln). Für jeden primitiven Typ gibt es eine korrespondierende Wrapper-Klasse, die den zugeordneten primitiven Typ repräsentiert, aber ein echter Objekttyp ist. Die Wrapper-Klasse für `int` ist beispielsweise `Integer`. Eine komplette Liste der primitiven Typen und ihrer Wrapper-Klassen findet sich in Anhang B.

Die folgende Anweisung „wickelt“ den Wert der Variablen `ix` vom primitiven Typ `int` in ein `Integer`-Objekt:

```
Integer iwrap = new Integer(ix);
```

Und nun kann `iwrap` beispielsweise in einer Sammlung vom Typ `ArrayList<Integer>` gespeichert werden. Das Speichern von primitiven Werten in Objektsammlungen wird jedoch noch mehr durch ein Sprachkonstrukt erleichtert, das als *Autoboxing* (übersetbar mit „automatischer Verpackung“) bezeichnet wird.

Immer dann, wenn der Wert eines primitiven Typs in einem Zusammenhang verwendet wird, der einen Wrapper-Typ erfordert, verpackt der Compiler den Wert des primitiven Typs automatisch in ein Objekt der passenden Wrapper-Klasse. Dies bedeutet, dass die Werte primitiver Typen direkt in eine Sammlung eingefügt werden können.

```
private ArrayList<Integer> notenListe;
...
public void speichereNoteInListe(int note)
{
    notenListe.add(note);
}
```

Die inverse Operation – das *Unboxing* – wird ebenfalls automatisch ausgeführt, sodass das Auslesen aus einer Sammlung etwa folgendermaßen aussehen könnte:

```
int ersteNote = notenListe.remove(0);
```

Autoboxing wird ebenfalls angewendet, wenn der Wert eines primitiven Typs als Parameter an eine Methode übergeben wird, die einen Wrapper-Typ erwartet, und wenn der Wert eines primitiven Typs in einer Variablen eines Wrapper-Typs gespeichert wird. In analoger Weise wird Unboxing angewendet, wenn der Wert eines Wrapper-Typs als Parameter an eine Methode übergeben wird, die einen primitiven Typ erwartet, und wenn er in einer Variablen mit einem primitiven Typ gespeichert wird. Beachten Sie, dass diese Verfahrensweise dazu führt, dass es fast so scheint, als ob man primitive Typen in Sammlungen speichern könnte. Doch der Typ der Sammlung muss immer noch unter Verwendung des Wrapper-Typs (z.B. `ArrayList<Integer>` und nicht `ArrayList<int>`) deklariert werden.

## Konzept

**Autoboxing**  
wird automatisch durchgeführt, wenn der Wert eines primitiven Typs in einem Kontext verwendet wird, in dem ein Wrapper-Typ erwartet wird.

### 6.10.1 Verwalten der Nutzungszählung

Die Kombination einer Map mit Autoboxing bietet eine einfache Möglichkeit, die Nutzungszählung von Objekten zu verwalten. Wir nehmen beispielsweise an, dass das Unternehmen, welches das System *Technischer-Kundendienst* verwendet, Beschwerden von seinen Kunden bekommt, weil einige der Antworten keinen Bezug zu den gestellten Fragen haben. Das Unternehmen könnte dann entscheiden, die in den Fragen verwendeten Wörter zu analysieren sowie weitere gezielte Antworten für diejenigen Wörter hinzuzufügen, die am häufigsten vorkommen und für die es keine fertigen Antworten gibt. Listing 6.8 zeigt einen Teil der Klasse `Wortzaehler`, die im Projekt *Technischer-Kundendienst-Analyse* zu finden ist.

**Listing 6.8**

Die Klasse `Wortzaehler` wird eingesetzt, um Worthäufigkeiten zu zählen.

```
import java.util.HashMap;
import java.util.HashSet;

/**
 * Aufzeichnungen, wie häufig jedes Wort vom Benutzer
 * eingegeben wurde.
 *
 * @author Michael Kolling und David J. Barnes
 * @version 1.0 (2016.02.29)
 */
public class Wortzaehler
{
    // Jedes Wort wird mit einer Zählung verbunden.
    private HashMap<String, Integer> zaehlungen;

    /**
     * Erzeuge einen Wortzähler.
     */
    public Wortzaehler()
    {
        zaehlungen = new HashMap<>();
    }

    /**
     * Aktualisiere die Zählungen aller Wörter in der Eingabe.
     * @param eingabe eine Menge von Wörtern, die vom Benutzer eingegeben wurden
     */
    public void woerterHinzufuegen(HashSet<String> eingabe)
    {
        for(String wort : eingabe) {
            int zaehler = zaehlungen.getOrDefault(wort, 0);
            zaehlungen.put(wort, zaehler + 1);
        }
    }
}
```

Die Methode `woerterHinzufuegen` erhält dieselbe Wortmenge, die dem `Beantworter` übergeben wird, sodass jedem Wort eine Häufigkeit zugeordnet werden kann. Die Häufigkeiten werden in einer Map gespeichert, die Zeichenketten auf ganzen Zahlen abbildet.

Beachten Sie den Einsatz der `HashMap`-Methode `getOrDefault`, die zwei Parameter besitzt: einen Schlüssel und einen vorgegebenen Wert. Wird der Schlüssel in der Map bereits verwendet, dann gibt diese Methode den entsprechenden Wert zurück. Falls aber der Schlüssel nicht verwendet wird, dann wird der vorgegebene Wert anstelle von `null` geliefert. Dadurch wird vermieden, zwei unterschiedliche Folgeaktionen zu schreiben, je nachdem, welcher Schlüssel in Benutzung ist und welcher nicht. Wenn wir stattdessen `get` verwenden würden, müssten wir so etwas wie Folgendes schreiben:

```
Integer zaehler = zaehlungen.get(wort);
if(zaehler == null) {
    zaehlungen.put(wort, 1);
}
else {
    zaehlungen.put(wort, zaehler + 1);
}
```

Beachten Sie, wie Autoboxing und Unboxing mehrere Male in diesen Beispielen eingesetzt werden.

**Übung 6.48** Was macht die Methode `putIAbsent` von `HashMap`?

**Übung 6.49** Fügen Sie eine Methode zur Klasse `Wortzaehler` im Projekt `Technischer-Kundendienst-Analyse` hinzu, um die Häufigkeit jedes Wortes auszugeben, nachdem die „ade“-Nachricht ausgegeben wurde.

**Übung 6.50** Geben Sie nur von denjenigen Wörtern die Häufigkeiten aus, die noch keine Schlüssel in `antwortMap` im der `Beantworter`-Klasse sind. Sie werden dazu eine sondierende Methode für `antwortMap` bereitstellen müssen.

## 6.11 Die Klassendokumentation schreiben

Während Sie an einem Projekt arbeiten, ist es wichtig, dass Sie neben der Entwicklung des Quelltextes auch die Dokumentation erstellen. Es ist leider nach wie vor noch allgemein üblich, dass Programmierer das Dokumentieren nicht sehr ernst nehmen, was später häufig zu ernsthaften Problemen führt.

Wenn Sie nicht genügend dokumentieren, dann kann es für einen anderen Programmierer (oder sogar für Sie selbst!) schwierig sein, Ihre Klassen zu verstehen. Typischerweise müssen Sie dann die Implementierung untersuchen, um Ihre Arbeitsweise zu verstehen. Das klappt eventuell noch in kleinen Studentenprojekten, aber in realen Projekten führt das zu erheblichen Schwierigkeiten.

Es ist bei kommerziellen Anwendungen durchaus üblich, dass der Quelltext aus mehreren Hunderttausend Zeilen in mehreren Tausend Klassen besteht. Stellen Sie sich vor, Sie müssten all das lesen, um die Arbeitsweise zu verstehen! Das kann eine einzelne Person nicht bewältigen.

Als wir Klassen wie `HashSet` oder `Random` aus der Java-Bibliothek benutzt haben, haben wir uns ausschließlich auf die Dokumentation verlassen, um ihre Dienstleistungen zu verstehen. Wir haben uns nicht ein einziges Mal die Implementierung dieser Klassen angesehen. Das hat funktioniert, weil diese Klassen einigermaßen gut dokumentiert sind (obwohl auch da noch Verbesserungen denkbar sind). Es wäre sehr viel schwieriger für uns geworden, wenn wir uns die Implementierungen der Klassen hätten ansehen müssen, bevor wir sie benutzen können.

In einem Entwicklungsteam für Software werden die Implementierungen der Klassen üblicherweise von mehreren Programmierern gemeinsam benutzt. Während Sie möglicherweise für die Klasse `Kundendienstsystem` aus unserem letzten Beispiel zuständig sind, implementiert ein anderer vielleicht die Klasse `Eingabesleser`. Es kann also vorkommen, dass Sie die eine Klasse implementieren und dabei Methoden von anderen Klassen aufrufen.

Das Argument, das bei Bibliotheksklassen so schlüssig ist, gilt in gleicher Weise für die Klassen, die Sie selbst erstellen: Wenn wir sie benutzen können, ohne dass wir ihre Implementierung lesen und verstehen müssen, dann wird uns das die Arbeit erleichtern. Wie bei den Bibliotheksklassen wollen wir nur die Schnittstelle einer Klasse sehen, nicht ihre Implementierung. Deshalb ist eine gute Klassendokumentation für Ihre eigenen Klassen ebenso wichtig.

### Konzept

Die **Dokumentation** einer Klasse sollte genau die Informationen bieten, die andere Programmierer benötigen, um die Klasse ohne einen Blick auf ihre Implementierung benutzen zu können.

Java-Systeme bieten ein Werkzeug namens **javadoc** an, mit dem solche Schnittstellenbeschreibungen aus Quelltexten generiert werden können. Die Dokumentation der Standardbibliothek beispielsweise, die wir uns angesehen haben, ist aus den Quelltexten der Klassen mithilfe von **javadoc** erzeugt worden.

### 6.11.1 javadoc in BlueJ nutzen

Die BlueJ-Umgebung benutzt **javadoc**, um für Ihre Klassen die Dokumentation auf zwei Wegen zu erzeugen:

- Sie können die Dokumentation für eine einzelne Klasse betrachten, indem Sie im aufklappbaren Listenfeld rechts oben im Editorfenster vom Eintrag **QUELLTEXT** zum Eintrag **DOKUMENTATION** wechseln (oder im Menü **WERKZEUGE** den Befehl **ANSICHT IMPLEMENTIERUNG/DOKUMENTATION** wählen).
- Sie können mit der Funktion **DOKUMENTATION ERZEUGEN** aus dem Menü **WERKZEUGE** die Dokumentation für alle Klassen im Projekt erzeugen.

Sie können weitere Details dazu im Tutorial für BlueJ nachlesen. Sie finden das Tutorial im Menü **HILFE**.

### 6.11.2 Die Elemente einer Klassendokumentation

Die Dokumentation einer Klasse sollte mindestens die folgenden Punkte enthalten:

- den Klassennamen
- einen Kommentar, der den allgemeinen Zweck und die Eigenschaften der Klasse beschreibt
- eine Versionsnummer
- den (oder die) Autorennamen
- eine Dokumentation für jeden Konstruktor und jede Methode

Die Dokumentation jedes Konstruktors und jeder Methode sollte enthalten:

- den Namen der Methode
- den Ergebnistyp
- die Namen und Typen der Parameter
- eine Beschreibung des Zwecks und der Arbeitsweise der Methode
- eine Beschreibung jedes Parameters
- eine Beschreibung des Ergebnisses

Zusätzlich sollte jedes Projekt einen Projektkommentar enthalten, der oft in einer „ReadMe“-Datei steht. In BlueJ ist dieser Projektkommentar über das Textblatt-Symbol in der linken oberen Ecke des Klassendiagramms zugreifbar.

**Übung 6.51** Benutzen Sie die Funktion **DOKUMENTATION ERZEUGEN** in BlueJ, um die Dokumentation für Ihr Kundendienstsystem zu erstellen. Untersuchen Sie das Ergebnis. Ist die Dokumentation korrekt? Ist sie vollständig? Welche Teile sind nützlich, welche eher nicht? Finden Sie Fehler in der Dokumentation?

Einige Elemente der Dokumentation, wie die Namen und die Parameter der Methoden, können immer aus dem Quelltext extrahiert werden. Andere Teile, wie beispielsweise die Kommentare für die Klasse, die Methoden und die Parameter, brauchen mehr Aufmerksamkeit, da sie leicht vergessen werden oder unvollständig oder inkorrekt sind.

In Java werden die Kommentare für **javadoc** mit einer speziellen Zeichenfolge eingeleitet:

```
/**  
 * Dies ist ein javadoc-Kommentar.  
 */
```

Die öffnenden Zeichen für einen Kommentar müssen zwei Sterne enthalten, damit sie von **javadoc** erkannt werden können. Wenn ein solcher Kommentar unmittelbar vor einer Klassendefinition steht, wird er als Klassenkommentar aufgefasst. Wenn er unmittelbar vor der Signatur einer Methode steht, wird er als ein Methodenkommentar aufgefasst.

Die genauen Details, wie die Dokumentation erzeugt und formatiert wird, sind für jede Programmiersprache und jede Programmierumgebung unterschiedlich. Der Inhalt sollte jedoch mehr oder weniger immer der gleiche sein.

Für Java in Verbindung mit **javadoc** stehen einige Schlüsselwörter zur Verfügung, mit denen die Dokumentation formatiert werden kann. Diese Schlüsselwörter beginnen immer mit dem Symbol @:

```
@version  
@author  
@param  
@return
```

**Übung 6.52** Finden Sie Beispiele für die Schlüsselwörter von **javadoc** im Quelltext des Projekts *Technischer-Kundendienst*. Wie beeinflussen diese die Formatierung der Dokumentation?

**Übung 6.53** Finden und beschreiben Sie weitere Schlüsselwörter von **javadoc**. Ein guter Platz zum Nachforschen ist die Online-Dokumentation der Java-Distribution von Sun Microsystems. Sie bietet ein Dokument mit dem Titel *javadoc – The Java API Documentation Generator* (beispielsweise unter <http://java.sun.com/javase/8/docs/technotes/tools/windows/javadoc.html>). In diesem Dokument werden die Schlüsselwörter **javadoc tags** genannt.

**Übung 6.54** Dokumentieren Sie alle Klassen in Ihrer Version des Kundendienstsystems in angemessener Weise.