PEARSON Education

ADDISON-WESLEY

MASTER CLASS

Karsten Samaschke

Java

Einstieg für Anspruchsvolle

ADDISON-WESLEY [in Kooperation mit] PEARSON Studium

3

Objektorientierte Programmierung

In diesem Kapitel werden wir eines der wichtigsten Themen bei der Entwicklung mit Java und auch anderen modernen Hochsprachen besprechen: Die Objektorientierung.

Objektorientierte Programmierung bedeutet einen Wandel in der Herangehensweise gegenüber traditionellen Verfahren – statt in Abläufen denkt man nun in Objekten. Eine Lösung – egal, ob eigenständiges Programm oder kleine Komponente – besteht dabei immer aus Objekten, die bestimmte Strukturen repräsentieren.

Das Geheimnis objektorientierter Programmierung besteht darin, zusammengehörige Daten und Methoden in einer eigenen Struktur zu kapseln. Dadurch werden gleich mehrere Ziele erreicht:

- Der Quellcode wird übersichtlicher, weil besser strukturiert
- Der Quellcode wird modularer, weil in einzelne kleine Bestandteile zerlegt
- Der Quellcode wird wiederverwendbarer, da einzelne Elemente in sich abgeschlossen sein können
- Der Quellcode wird sicherer, denn in kleineren Elementen und Strukturen kann man Fehler besser erkennen
- Einzelne oder alle Komponenten einer Applikation können auf Wunsch in anderen Applikationen weiter verwendet werden

Wie Sie diese Ansprüche verwirklichen können, werden wir im Folgenden betrachten.

3.1 Klassen und Objektinstanzen

„Objektorientierung“ ist eines der sogenannten Buzz-Words der 90er Jahre gewesen: Alles wurde auf einmal objektorientiert, viele Sprachen wurden mit objektorientierten Ansätzen versehen – meist nicht unbedingt zu ihrem Besten, denn eine nachträglich aufgepfropfte Objektorientierung leidet oftmals unter

Einschränkungen und Limitierungen, die die alltägliche Arbeit unnötig verkomplizieren.

Die gute Nachricht: Bei Java ist das anders. Java ist von vornherein als objektorientierte Sprache entwickelt worden. Bei Java ist Objektorientierung nicht nachträglich aufgesetzt, sondern integraler Bestandteil der Sprache. Bis auf die primitiven Datentypen ist bei Java alles ein Objekt.

Die schlechte Nachricht: Dies verhindert nicht, daß wir uns intensiver mit dieser Materie auseinandersetzen müssen. Oftmals haben Ein- und Umsteiger ein Denkproblem, wenn sie plötzlich objektorientiert arbeiten sollen.

3.1.1 Abstraktion

Die wichtigste Idee der objektorientierten Programmierung ist die Trennung von *Konzept* und *Umsetzung*. Dies bedeutet, das ein fundamentaler Unterschied zwischen einer *Klasse* und einem *Objekt* existiert: Eine Klasse beschreibt eines oder mehrerer Objekte, während das Objekt eine konkrete Manifestierung einer Klasse darstellt.

Nehmen wir ein Beispiel. Denken Sie generell an Autos – nicht an ein bestimmtes Fabrikat oder ein bestimmtes Modell, sondern an das, was ein Auto ausmacht: Autos bestehen aus Reifen, Türen und einem Motor. Sie verfügen über eine Farbe und haben einen Kilometerstand, der angibt, wie weit das Fahrzeug schon bewegt worden ist. Autos können fahren und dabei eine bestimmte Anzahl an Personen transportieren. All dies sind Eigenschaften und Fähigkeiten aller Autos – nicht eines konkreten Fahrzeuges. Derartige Eigenschaften und Fähigkeiten (wir sprechen hier allerdings eher von Methoden) können in einer Klasse definiert werden.

Klasse

Eine *Klasse* definiert Eigenschaften und Methoden. Eine Klasse ist etwas Abstraktes – sie erlaubt es uns, etwas zu beschreiben und ähnelt dabei sehr stark einem Bauplan, einem Konzept oder einem Rezept.

Klassen beschreiben konkrete Elemente als Summe ihrer *Eigenschaften*. Eigenschaften repräsentieren Zustände und Werte, führen aber selbst keine Verarbeitung durch. Eine Klasse, die Autos beschreiben sollte, könnte demnach etwa über die Eigenschaften „Farbe“, „Alter“ oder „Kilometerstand“ verfügen. Eigenschaften repräsentieren somit Zustände eines Objekts.

Daneben verfügt eine Klasse meist auch über *Methoden*. Diese Methoden können auf die Eigenschaften der Klasse wirken oder Aktionen durchführen, die andere Klasse beeinflussen. Eine Auto-Klasse könnte etwa über die Methoden „Altere“, „Erhöhe die Anzahl der Beulen“, „Verdrecke die Sitze“ oder „Drehe den Tacho zurück“ verfügen. Methoden repräsentieren also ein Verhalten eines Objekts.

Zuletzt gibt uns eine Klasse auch immer Aufschluß darüber, wie wir ein konkretes Objekt erzeugen können.

Eine Klasse stellt uns also folgende Dinge zur Verfügung:

- Eigenschaften, die das konkrete Objekt beschreiben
- Methoden, mit denen diese Eigenschaften manipuliert werden können
- Informationen darüber, wie konkrete Objekte erzeugt oder hergestellt werden können

Objekt

Mit Hilfe der von einer Klasse definierten Eigenschaften, Methoden und Informationen ist jedes konkrete *Objekt* beschreibbar. Wir sind somit in der Lage, eine beliebige Anzahl an konkreten Objekten zu erzeugen. Jedes dieser Objekte steht für sich, beschreibt sich selbst und ist dadurch von anderen Objekten unterscheidbar. Ein konkretes Objekt ist dabei jedoch immer eine Instanz der Klasse, nach der es modelliert worden ist.

Nehmen wir eine Klasse „Auto“: Diese Klasse stellt die Eigenschaften bereit, die wir benötigen, um jedes derzeit auf einer Autobahn befindliche Fahrzeug hinlänglich zu beschreiben. Wir sind darüber hinaus in der Lage, neue Fahrzeuge zu erstellen (oder zu bauen) und können mit Hilfe der in der Klasse definierten Methoden die Eigenschaften von Fahrzeugen manipulieren..

Sie werden oftmals feststellen, daß die Begriffe „Instanz“ und „Objekt“ synonym gebraucht werden. Tatsächlich werden Sie das hier in diesem Buch auch feststellen können. Natürlich bestehen im Sinne einer Definition Unterschiede zwischen Instanzen und Objekten. Praktisch spielt dies aber keine Rolle und die Begriffe werden so eingesetzt, wie es besser klingt. Und genau so halten wir es in diesem Buch auch.

Durch die Unterscheidung von Klassen und Objekten können wir uns auf das Wesentliche konzentrieren und bei der Definition einer Klasse die unwichtigen Details außen vor lassen. Wir können somit das Wesen eines Problems leichter erfassen und reduzieren im Idealfall den Aufwand zur Problemlösung enorm, da wir Methoden und Eigenschaften einmal definieren und mehrfach wiederverwenden können.

3.1.2 Kapselung

Klassen definieren Eigenschaften und Methoden. Die Eigenschaften werden durch Variablen repräsentiert, die für jede Instanz neu angelegt und instanziert werden. Diese Variablen werden auch gerne als *Attribute*, *Member-* oder *Instanzvariablen* bezeichnet und repräsentieren den Zustand des beschriebenen Objekts.

Methoden repräsentieren dagegen das Verhalten eines Objekts. Sie erlauben das Manipulieren der Zustände der aktuellen Objektinstanz oder anderer Instanzen.

Klassen fassen Methoden und Eigenschaften zusammen, was als *Kapselung* bezeichnet wird. Die Kapselung dient dem Zweck, die Komplexität bei der Bedienung eines Objekts zu verringern – wir müssen nicht speziell wissen, wie ein Auto im Inneren aufgebaut ist, um es auf einer Autobahn fahren zu können. Die Kapselung sorgt dafür, das wir von diesen Informationen ferngehalten werden. Angenehmer Nebeneffekt: Wir können bei der Arbeit mit einem konkreten Objekt meist nicht all zu viel Schaden anrichten, denn wir erhalten aufgrund der Kapselung keinen Zugriff auf die Membervariablen und die Hilfsmethoden des Objekts.

3.1.3 Wiederverwendbarkeit

Aufgrund der Kapselung von Code steigt die *Wiederverwendbarkeit* dieses Codes. Dies bedeutet für uns, das wir die in einer Klasse definierten Methoden auch in den konkreten Objekten verwenden können, ohne sie neu implementieren zu müssen. Ebenso ist es möglich, in Klassen definierten Code aus anderen Methoden oder Objekten heraus zu verwenden. Dies alles ermöglicht es, Code wieder zu verwenden, ohne ihn stets neu schreiben zu müssen. Dadurch steigt die Effizienz bei der Entwicklung, insbesondere deshalb, weil die Fehleranfälligkeit schon beim Schreiben von Applikationen sinkt.

3.2 Beziehungen zwischen Klassen

Klassen können auf unterschiedliche Art und Weise miteinander in Beziehung stehen. Bei der objektorientierten Programmierung unterscheiden wir drei Beziehungstypen:

- Spezialisierung
- Komposition
- Assoziation

3.2.1 Spezialisierung

Die *Spezialisierung* von Klassen soll an einem einfachen Beispiel erläutert werden: Autos, Lastwagen, Lokomotiven, Straßenbahnen, Busse, Motorräder und Fahrräder haben alle eines gemeinsam: Sie sind Fahrzeuge. Wenn wir die Gemeinsamkeiten aller dieser Fahrzeuge untersuchen, läßt sich feststellen, das sie sich in einigen Punkten ähnlich sind:

- Sie verfügen über die Fähigkeit, sich zu bewegen
- Sie verfügen über Räder
- Sie verfügen über ein Steuer-Instrument (Lenker, Lenkrad, Joystick oder ähnliches)
- Sie verbrauchen Energie
- Sie haben eine wie auch immer geartete Art von Karosserie (und sei es nur der Fahrrad-Rahmen)
- Sie haben eine Farbe

Wenn wir alle diese Fahrzeuge beschreiben sollen, würden wir sinnvollerweise eine Klasse `Fahrzeug` schreiben, die alle oben genannten Eigenschaften und Methoden beinhaltet.

Alle oben genannten Fahrzeuge verfügen über die Eigenschaften und Methoden der Klasse `Fahrzeug` und fügen dieser eigene Methoden und Eigenschaften hinzu – ein Auto würde beispielsweise die Art des Karosserie-Aufbaus definieren, wogegen eine Fahrrad vielleicht eher darüber definiert wird, ob es ein Sportfahrrad oder ein Mountainbike ist. Eine Lokomotive benötigt Informationen darüber, ob sie einen Elektro-, Diesel- oder Dampfantrieb hat.

Allgemeiner gesagt: Die Spezialisierung beruht darauf, das eine Klasse B Eigenschaften einer Klasse A besitzt und dieser eigene Eigenschaften oder Methoden hinzufügen kann. Diese Art der Beziehung wird bei Java durch *Ableitung* oder *Vererbung* ausgedrückt. Diese Vererbung kann dabei mehrstufig sein – eine Klasse kann von einer anderen Klasse erben, die ihrerseits auch von einer Klasse geerbt hat. Dadurch entsteht eine Vererbungshierarchie, die für Fahrzeuge beispielsweise so aussehen könnte:

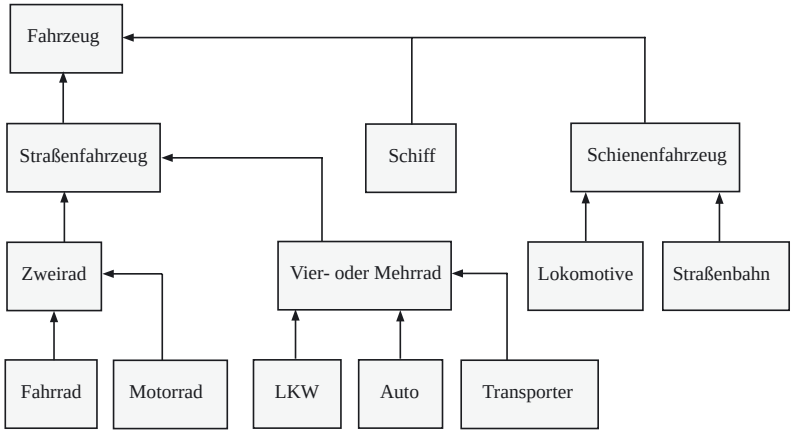

Abbildung 3.1
Vererbungshierarchie

Im Vergleich zu anderen Programmiersprachen beschränkt Java die Möglichkeiten der Vererbung: Klassen können in Java nur von einer Super-Klasse erben – ein paralleles Erben von mehreren Super-Klassen ist nicht möglich, kann aber mit Hilfe von Interfaces nachgestellt werden.

3.2.2 Komposition und Aggregation

Eine andere Art von Beziehung besteht zwischen einem Auto und seinen Komponenten: So verfügt ein Auto in der Regel über vier oder fünf Reifen, die ihrerseits sinnvollerweise durch eine Klasse `Tyre` repräsentiert werden könnten. Das Auto verfügt dann beispielsweise über ein Array aus `Tyre`-Instanzen. Ebenfalls könnte ein Auto über ein Lenkrad verfügen, das durch eine `Wheel`-Instanz repräsentiert werden würde. Diese Zusammensetzung eines Objekts aus anderen Objekten wird als *Komposition* bezeichnet.

Ebenfalls im Zusammenhang mit Komposition ist oftmals von *Aggregation* die Rede. Dieser Begriff bezeichnet eine nicht-existenzielle Beziehung zwischen zwei Objekten. Ein Beispiel für die Aggregation wäre das Hinzufügen von Koffern zum Kofferraum eines Fahrzeuges – das Fahrzeug ist auch funktionsfähig, wenn keine Koffer eingeladen worden sind. Anders ist dies bei Rädern der Fall – ohne Räder funktioniert ein Fahrzeug in der Regel nicht.

Diese Art von Unterscheidung zwischen Aggregation und Komposition ist mehr inhaltlicher denn praktischer Natur – in der Implementierung gibt es in der Regel keinen Unterschied, da beide Beziehungen über Instanz-Variablen abgebildet werden. Es obligt der Klasse selber, die Abhängigkeiten zu prüfen und gegebenenfalls entsprechende Warnungen oder Fehler zu produzieren.

3.2.3 Assoziation

Die *Assoziation* ist die allgemeinste Art von Beziehungen zwischen Klassen. Sie beruht darauf, das eine Klasse eine Instanz einer anderen Klasse samt deren Methoden nutzt. Ein Objekt A nutzt also Methoden eines Objekts B für seine Zwecke aus. Ein typischer Anwendungsfall sind Hilfsklassen, die Methoden beinhalten, die von anderen Klassen gleichermaßen verwendet werden sollen.

Ebenfalls eine Verwendung von Assoziationen ist der Einsatz von Instanzvariablen, die Referenzen auf andere Objekttypen halten oder die Verwendung von Objekten als Methoden-Argumenten.

3.3 Polymorphie

Der Begriff *Polymorphie* bezeichnet die Fähigkeit von Objektvariablen, Objekte unterschiedlicher Typen aufzunehmen. Dies ist allerdings nicht unbegrenzt möglich, sondern ist auf Objekte eines Typs und davon abgeleitete Objekte beschränkt. Ein Beispiel: Eine Variable vom Typ `Fahrzeug` kann sowohl `Fahrzeug`-, als auch `Boot`-, `Zweirad`- oder `Auto`-Objekte halten. Dagegen kann eine Variable vom Typ `Zweirad` nur `Zweirad`-, `Fahrrad`- und `Motorrad`-Objekte halten – bei Zuweisung eines `Auto`-Objekts würde ein Fehler auftreten. Die Polymorphie funktioniert also nur innerhalb einer Vererbungshierarchie (und dort auch nur abwärts) und spiegelt dabei diese Hierarchie wieder.

Spannend wird Polymorphie dann, wenn mit *überlagerten* Methoden gearbeitet wird. Das Prinzip dahinter: In einer übergeordneten Klasse A wird eine Methode b definiert. Diese kann nun von einer ableitenden Klasse C überschrieben werden. Wird nun eine Instanz der Klasse C mit der überschriebenen Methode b einer Variablen vom Typ A zugewiesen – was dank Polymorphie funktioniert, da C von A erbt und somit in der Hierarchie unterhalb von A liegt – wird erst zur Laufzeit bestimmt, welche Ausprägung von b tatsächlich eingebunden wird: Die von A definierte Version, oder die von C definierte. Sollte eine Variable vom Typ C zugewiesen sein, wird die von C definierte Version von b eingebunden.

Nehmen wir nun an, eine Klasse D erbt ebenfalls von A und definiert keine eigene Version der Methode b. Was wird nun geschehen? Wird ein Fehler generiert? Nein. Zur Laufzeit wird die von der nächsthöheren Hierarchie-Ebene definierte Version der Methode b eingebunden. Hat diese ebenfalls keine eigene Definition der Methode, wird die Methode b von deren Basis-Klasse eingebunden.

Ein Beispiel: In der Klasse `Fahrzeug` definieren wir eine Methode `fahre()`, die dafür sorgt, das sich das Fahrzeug (virtuell gesehen) bewegt. Dabei wird der Kilometerstand erhöht und der Tankinhalt verringert. Die Klasse `Schienenfahrzeug`, die als Basis-Klasse für `Lokomotive` und `Straßenbahn` fungiert, muss diese Methode anders implementieren – Loks und Straßenbahnen verfügen in der Regel über einen Elektro-Antrieb und leeren deshalb keinen Tank. Die Klasse `Straßenfahrzeug` dagegen muss diese Methode nicht anders implementieren und kann die ursprüngliche Version verwenden.

Weisen wir nun einer Variablen vom Typ `Fahrzeug` eine `Schienenfahrzeug`-Instanz zu und rufen deren Methode `fahre()` auf, wird zur Laufzeit der Applikation die `fahre()`-Implementierung der `Schienenfahrzeug`-Klasse verwendet. Selbiges gilt für Instanzen der `Lokomotive`- und `Straßenbahn`-Klassen, soweit diese nicht selbst wieder über eine eigene Implementierung (etwa für eine dieselbetriebene Lok) verfügen. Wird nun aber eine Instanz der `Auto`-Klasse zugewiesen, die keine eigene Überlagerung von `fahre()` hat, überprüft der Java-Interpreter auch die Super-Klasse `Vier- oder Mehrrad` auf das Vorhandensein einer `fahre()`-Überlagerung. Dieser Prozess wiederholt sich für die jeweilige Super-Klasse solange, bis eine fahre()-Implementierung gefunden worden ist. Sollte dies nicht geschehen – etwa weil es keine weitere Überlagerung der Methode gibt – wird die Implementierung der Basis-Klasse `Fahrzeug` eingebunden werden.

Dieses Verhalten führt in der Praxis zu sehr eleganten Lösungen, da Entwickler so nicht gezwungen sind, stets alle Methoden einer Super-Klasse zu implementieren. Stattdessen können sie sich in Ableitungen auf die Methoden beschränken, die sie tatsächlich anders oder neu implementieren müssen, um die gewünschte Funktionalität sicher zu stellen.

3.4 Definition von Klassen

Klassen zu definieren ist nicht schwer. Alles, was wir benötigen, ist das Schlüsselwort *class*, gefolgt von dem Klassen-Namen. Der Aufbau einer Klassen-Deklaration sieht also immer so aus:

```
[<Zugriffmodifier>] class <Klassen-Name> {
   ...
}
```

Beginn und Ende der Klasse werden dabei durch geschweifte Klammern gekennzeichnet. Wollten wir also eine Klasse `SimpleCar` deklarieren, könnte ein einfacher Ansatz so aussehen:

```
public class SimpleCar {}
```

Achtung

Java hat eine goldene Regel: Jede öffentliche Klasse gehört in eine eigene gleichnamige **.java*-Datei. Dies bedeutet, daß eine öffentliche Klasse *SimpleCar* in einer Datei *SimpleCar.java* gespeichert wird. Weitere Klassen darf diese Datei nur aufnehmen, wenn diese innerhalb von *SimpleCar* geschachtelt werden oder nicht als öffentlich deklariert sind, anderenfalls müssen die entsprechenden Klassen auch wieder innerhalb einer eigenen Datei definiert werden.

3.4.1 Eigenschaften / Instanz-Variablen

Eigenschaften werden in der Regel über *Getter*- und *Setter*-Methoden implementiert. Innerhalb dieser Methoden wird auf Instanz-Variablen zugegriffen, die die verschiedenen Werte halten.

In unserer Auto-Klasse werden wir nun drei Instanz-Variablen definieren, die den verschiedenen Eigenschaften eines einfachen Autos entsprechen sollen:

Listing 3.1
Eine einfache Klasse

```
public class SimpleCar {

   private String manufacturer = "Unknown";
   private String color = "White";
   private double miles = 0;

}
```

Diese internen Variablen sind nicht von außen (also anderen Klassen oder auch anderen Objekt-Instanzen) sichtbar. Deklariert werden Sie genauso, wie Sie auch normale Variablen in Methoden deklarieren, also in der Form

```
[<Zugriffsmodifier>] <Typ> <Name> [= <Standardwert>];
```

Um sie erreichbar zu machen, nutzen wir Getter- und Setter-Methoden.

3.4.2 Getter und Setter

Mit Hilfe von Getter-Methoden, kann der Wert der repräsentierten Variable abgerufen werden. Eine Getter-Methode hat in der Regel diesen Aufbau:

```
[<Zugriffsmodifier>] <Typ> get<Variable>() {
   return <Variable>;
}
```

Der Zugriffsmodifizierer ist optional. Wird er nicht angegeben, geht Java davon aus, daß die Methode zwar öffentlich erreichbar sein soll, sich dies aber auf das aktuelle Package beschränkt (vgl. *public*-Modifier). Vor der Rückgabe eines Wertes kann der Setter noch interne Überprüfungen vornehmen. So kann sichergestellt werden, dass nur gültige Werte an die übergeordnete Methode zurückgegeben werden. Angenehmer Nebeneffekt: Es muss kein Zugriff auf die internen Variablen der Klasse gewährt werden – weder lesend noch schreibend.

Das Gegenstück zu Getter-Methoden sind Setter-Methoden. Sie erlauben eine Zuweisung von Werten zu den repräsentierten Variablen und nehmen stets einen Parameter entgegen. Eine Setter-Methode sieht in der Regel so aus:

```
[Zugriffsmodifier] void set<Variable>(<Typ> <Name>) {
   this.<Variable> = <Name>;
}
```

Eine Setter-Methode erlaubt es, vor dem Zuweisen des Wertes an die interne Variable noch Überprüfungen vorzunehmen. Die Klasse kapselt damit ihren internen Aufbau und stellt sicher, das nur gültige Werte zugewiesen werden können. Dadurch steigt die Sicherheit der Applikation und die Fehleranfälligkeit verringert sich.

Übertragen auf unsere Klasse ergibt sich folgender Aufbau:

Listing 3.2
Getter und Setter einer Klasse

```
public class SimpleCar {

   private String manufacturer = "Unknown";
   private String color = "White";
   private double miles = 0;

   public String getManufacturer() {
      return manufacturer;
   }

   public void setManufacturer(String manufacturer) {
      this.manufacturer = manufacturer;
   }

   public String getColor() {
      return color;
   }

   public void setColor(String color) {
      this.color = color;
   }

   public double getMiles() {
      return miles;
   }

   public void setMiles(double miles) {
      this.miles = miles;
   }
}
```

Im Grunde ist damit unsere Klasse bereits fertig – sie kapselt die einzelnen Eigenschaften eines (noch) abstrakten Autos. In der derzeitigen Form haben wir eine reine Datenklasse geschrieben – oftmals ist dies genau das Richtige, um eine Repräsentation von zusammengehörigen Daten zu erreichen.

Jedoch werden Sie oftmals keine reinen Datenklassen einsetzen – meist verfügen die Klassen auch noch über Methoden, mit deren Hilfe die enthaltenen Daten manipuliert oder bestimmte Informationen abgerufen werden können. Dinge eigentlich, die zwar meist auch von außen geschehen könnten, dann aber eben nicht Bestandteil des Objekts wären, und für jedes andere Projekt neu implementiert werden müssten, was inbesondere im Hinblick auf Wartbarkeit nicht optimal und befriedigend wäre.

Also fügen wir einige Methoden zur Klasse hinzu. Die Methode `drive()` erlaubt es uns, eine Streckenangabe in Kilometern zu übergeben und berechnet diese Entfernung in Meilen um. Mit Hilfe der Methode `getKilometers()` erhalten wir die Möglichkeit, eine Meilen-Angabe in Kilometer umzurechnen. Die Methode `getStatus()` schließlich gibt uns den Status des PKW anhand der gefahrenen Meilen zurück:

Der Unterschied zwischen *Eigenschaften* (*Properties*) und *Methoden* (*Funktionen*) ist bei Java mehr geistiger als syntaktischer Natur. Tatsächlich werden mit Properties die Methoden bezeichnet, die als Getter oder Setter für Instanz-Variablen fungieren, während Methoden eine verarbeitende Funktion innerhaben und Werte manipulieren oder Operationen durchführen. Lassen Sie sich also nicht verwirren – akzeptieren Sie die Begriffe „Eigenschaft“ oder „Property“ als Synonym für Getter und Setter. In der Praxis wird bei Java ohnehin mehr von Getter- und Setter-Methoden gesprochen, als von Objekt-Eigenschaften.

Listing 3.3 Repräsentiert ein Fahrzeug: SimpleCar.java

```
public class SimpleCar {

   private String manufacturer = "Unknown";
   private String color = "White";
   private double miles = 0;

   private final double KMTOMILE = 0.62137;
   private final double MILETOKM = 1.60934;

   public String getManufacturer() {
      return manufacturer;
   }

   public void setManufacturer(String manufacturer) {
      this.manufacturer = manufacturer;
   }

   public String getColor() {
      return color;
   }

   public void setColor(String color) {
      this.color = color;
   }

   public double getMiles() {
      return miles;
   }

   public void setMiles(double miles) {
      this.miles = miles;
   }

   public void drive(double kilometres) {
      this.miles += Math.round(
         (kilometres * KMTOMILE)* 100.) / 100.;
   }

   public double getKilometres(long miles) {
      return Math.round((miles * MILETOKM) * 100.) / 100.;
   }

   public String carStatus() {
      String status = "old and used";
      if(this.getMiles() <= 10) {
         status = "absolutely new";
      }
      else if(this.getMiles() <= 1000) {
         status = "new";
      }
      else if(this.getMiles() <= 10000) {
         status = "nearly new";
      } else if(this.getMiles() <= 20000) {
         status = "in perfect shape";
      }
      else if(this.getMiles() <= 50000) {
         status = "used car";
      }
```

```
        else if(this.getMiles() <= 150000) {
            status = "fair offer";
        }

        return status;
    }
}
```

Listing 3.3 *(Forts.)*
Repräsentiert ein Fahrzeug: SimpleCar.java

Direkt unterhalb der drei Instanz-Variablen haben wir noch die beiden konstanten Variablen `KMTOMILE` und `MILETOKM` hinterlegt, die uns die von den Methoden `drive()` und `getKilometres()` vorzunehmenden Umrechnungen erleichtern, indem sie die Umrechnungsfaktoren beinhalten.

Die Methode `drive()` simuliert eine Auto-Fahrt, deren Entfernung in Meilen angegeben wird, `getKilometres()` dient der generellen Umrechung einer Meilen-Angabe in Kilometer. Bei beiden Umrechnungen wird auf zwei Stellen nach dem Komma gerundet.

Die Methode `carStatus()` liefert eine Status-Angabe über den Zustand des Autos – zwar nur anhand der Kilometer geschätzt, aber sicherlich zutreffender als die Angaben des manches Gebrauchtwagen-Händlers.

Damit ist unsere Klasse fertig und einsatzbereit. Sehen wir uns im Folgenden an, wie wir Instanzen der Klasse erzeugen und mit diesen arbeiten können.

3.5 Erzeugen von Klassen-Instanzen

Klassen-Instanzen werden mit Hilfe des Schlüsselworts *new* erzeugt. Die Syntax dafür sieht folgendermaßen aus:

```
<Typ> <Instanz> = new <Typ>();
```

Kommt Ihnen das bekannt vor? Falls ja, haben Sie gut beobachtet: Die Syntax ist ähnlich der Syntax, mit der Sie schon bisher Variablen deklariert haben. Tatsächlich haben Sie dort teilweise auch schon Klassen-Instanzen erzeugt, nur eben nicht von selbstdefinierten Klassen, sondern von Klassen, die die Java-Runtime bereitstellt.

Übertragen wir diese Syntax auf unsere `SimpleCar`-Klasse, dann könnten wir folgenden Code schreiben:

```
public class SimpleCarInvoker
{

    public static void main(String[] args) {
        SimpleCar car = new SimpleCar();
    }
}
```

Wir haben zwar nunmehr eine neue Instanz der `SimpleCar`-Klasse erzeugt, so richtig sinnvoll mit ihr gearbeitet haben wir aber noch nicht.

Wenn Sie eine neue Instanz einer Klasse erzeugen wollen, die sich innerhalb einer anderen Klasse befindet, dann können Sie dies aus der äußeren Klasse heraus so wie gewohnt machen. Was aber, wenn Sie eine neue Instanz der inneren Klasse erzeugen wollten?

Dann müssten Sie dem Typ immer den Typ der äußeren Klasse voranstellen:

```
<äußerer Typ> <Instanz> =
    new <äußerer Typ>();
```

Bei einer äußeren Klasse `WrapperClass` und einer inneren Klasse `ContainedClass` sähe dies dann so aus:

```
WrapperClass.ContainedClass cls = new WrapperClass.ContainedClass();
```

Selbstverständlich können Sie auch die äußere Klasse unter Verwendung des `using`-Statements einbinden. Dann sähe der Aufbau so aus:

```
using WrapperClass;
...
ContainedClass myInstance
= new ContainedClass();
```

Listing 3.4
Erzeugen einer neuen Instanz der SimpleCar-Klasse (SimpleCarInvoker.java)

3.6 Qualifizierung: Zugriff auf Methoden

Der Zugriff auf Methoden und Eigenschaften einer Klasse erfolgt in Form der sogenannten *Punkt-Notation*, die auch als *Qualifizierung* bezeichnet wird. Diese trennt Instanzvariable und deren Methode durch einen Punkt:

```
<Rückgabe> = <Variable>.<Methode>;
```

Wollten wir also unserer `SimpleCar`-Klasseninstanz Werte zuweisen oder von ihr Daten abrufen, könnte dies folgendermaßen aussehen:

Listing 3.5 Zuweisen und Abrufen von Daten einer Klasse

```
public class SimpleCarInvoker
{
   public static void main(String[] args) {
      SimpleCar car = new SimpleCar();
      car.setManufacturer("BMW");
      car.setColor("Blue");
      car.drive(1000);

      System.out.println(String.format(
         "You are driving a %s with %s color which " +
         "has the following status: %s",
             car.getManufacturer(),
             car.getColor(),
             car.carStatus()));
   }
}
```

Nun haben wir eine funktionsfähige Applikation. Gestartet wird sie über Kommandozeile mit folgendem Befehl:

```
java SimpleCarInvoker
```

Wenn wir sie ausführen, werden wir folgende Ausgabe erhalten:

You are driving a BMW with Blue color which has the following status: new

Soweit ist unsere Klasse gut funktionsfähig. Gehen wir nun an die Arbeit, sie ein wenig zu verfeinern.

3.7 Konstruktoren und Destruktoren

Konstruktoren und *Destruktoren* erfüllen verschiedene und völlig gegensätzliche Zwecke – sind jedoch beides wichtige Bestandteile von Java-Klassen: Konstruktoren initialisieren eine Klasse und letztere räumen hinterher wieder auf. Der Zeitpunkt, an dem Konstruktoren wirksam werden, läßt sich ziemlich genau bestimmen: Sobald eine neue Klassen-Instanz angelegt wird, erfolgt die Einbindung des Konstruktors.

Bei Destruktoren dagegen ist der Zeitpunkt nicht so genau bestimmbar: Sie werden aufgerufen, wenn die Klassen-Instanz vom Java-eigenen *Garbage-Collector* (quasi dem Müllmann) beseitigt wird. Der genaue Zeitpunkt ist durch das *Garbage-Collector*-Konzept nicht genau bestimmbar – beruht dieser Ansatz des Aufräumens doch darauf, dass er asynchron erfolgt.

3.7.1 Konstruktor

Syntaktisch ist der Aufbau eines Konstruktors einfach gehalten – es handelt sich um eine Methode, die den gleichen Namen wie die Klasse trägt:

```
<Klassenname>(<Parameter>) { ... }
```

Konstruktoren verfügen über folgende wesentliche Eigenschaften:

- Sie heißen wie die umgebende Klasse
- Sie haben keinen Rückgabetyp
- Sie sind nicht direkt aufgerufen werden
- Sie können keine Werte zurückgeben

Im Lebenszyklus einer Klasse kommt der Konstuktor an sehr exponierter und zeitiger Stelle: Direkt nachdem Java den Speicherplatz für das neue Objekt reserviert und die internen Variablen initialisiert hat, wird der Konstruktor ausgeführt – also noch vor allen Methoden und Zuweisungen.

Wenn wir in unserer `SimpleCar`-Klasse einen Konstruktor verwenden würden, könnte dieser folgenden Aufbau haben:

Listing 3.6
Konstruktor einer Klasse

```
public class SimpleCar {

   private String manufacturer = "Unknown";
   private String color = "White";
   private double miles = 0;

   private final double KMTOMILE = 0.62137;
   private final double MILETOKM = 1.60934;

   /**
    * Constructor for the class
    */
   public SimpleCar() {
      this.setManufacturer("Porsche");
      this.setColor("Blue");
   }

   ...
}
```

Innerhalb des Konstruktors werden Hersteller und Farbe mit Hilfe der Setter-Methoden der Klasse zugewiesen. Sollten Sie nun direkt nach der Instanzierung der Klasseninstanz per

```
SimpleCar myCar = new SimpleCar();
```

die entsprechenden Werte für Hersteller und Farbe abrufen, würden Sie erfahren, dass Sie einen blauen Audi besitzen.

3.7.2 Destruktor

Wie bereits zuvor erwähnt, dienen Destruktoren dem Aufräumen, wenn eine Klasseninstanz nicht mehr benötigt wird. Sie könnten einen Destruktor nutzen, um Datenbank-Verbindungen wieder zu schließen oder andere, möglicherweise komplexere, Aufräumvorgänge durchzuführen.

Die Syntax eines Destruktors ist sehr einfach: Es handelt sich immer um die Methode `finalize`, die im Gegensatz zu einem Konstruktor übrigens auch direkt aus anderen Methoden heraus aufgerufen werden kann:

```
protected void finalize() throws Throwable { ... }
```

Innerhalb der Methode, die die von `Object` geerbte Methode `finalize` überschreibt, könnten Sie nun Ihren Code zum Aufräumen bereitstellen. Es bietet sich an der Stelle übrigens an, bei eigenen `finalize`-Implementierungen zumindest noch den Aufruf des Super-Destruktors einzufügen. Daher sollte die einfachste `finalize`-Implementierung so aussehen:

```
protected void finalize() throws Throwable {
   super.finalize();
}
```

Sie sollten übrigens nicht darauf warten, daß die Methode vom System aufgerufen wird. Das kann innerhalb kurzer Zeit passieren, muss es aber nicht – dies entscheidet der Garbage-Collektor. Aus diesem Grund empfiehlt es sich, die `finalize`-Methode bei Bedarf manuell einzubinden, um eventuell nötige Aufräumarbeiten zeitnah vornehmen zu können.

3.8 Methoden-Überladung

Haben Sie sich nicht auch schon beim Erstellen von Methoden gefragt, warum Sie die Parameter immer so starr vorgeben müssen und nicht mit einer Art von optionalen Parametern arbeiten können?

Oder anders herum: Sie möchten mehrere Methoden anbieten, die allesamt das gleiche erledigen, aber unterschiedliche Parameter besitzen? Dann hatten Sie bisher ein echtes Problem – schließlich durften Sie derartige Methoden immer wieder neu implementieren, auch wenn sich nur der Kopf unterschieden hat.

Aus diesem Grund bietet Java ein Feature, das alle oben genannten Probleme löst und Ihnen das Leben deutlich erleichtert: *Methoden-Überladung* – gerne auch Neudeutsch als *Method-Overloading* bezeichnet.

Unter Methoden-Überladung versteht man die Deklaration von Methoden innerhalb einer Klassen-Hierarchie, die zwar den gleichen Namen und den gleichen Rückgabewert haben, sich dennoch in Rückgabe-Typ und/oder Anzahl bzw. Typ der Parameter unterscheiden.

Intern geschieht beim Aufruf einer überladenen Methode folgendes: Der Java-Interpreter überprüft, ob es für den angegebenen Methodennamen eine Imple-

mentierung gibt, die genau die angegebenen Objekt-Typen (oder deren Basis-Typen) in der angegebenen Reihenfolge beinhaltet. Sollte er eine derartige Übereinstimmung finden, wird die entsprechende Implementierung der Methode ausgeführt.

Achtung

Bei der Überladung von Methoden spielt der Name des jeweiligen Parameters für den Interpreter keinerlei Rolle. Relevant sind lediglich die Typen der Parameter und deren Reihenfolge. Eine Methode

```
void doSomething(String myParam) {}
```

ist also für den Interpreter (und auch schon für den Compiler) äquivalent zu einer Methode

```
void doSomething(String myOtherParam) {}
```

In der Regel sollten Sie bereits beim Kompilieren des Quelltextes zu Java-Bytecode eine entsprechende Fehlermeldung erhalten.

Übertragen wir dieses Prinzip auf die `SimpleCar`-Klasse, bietet sich eine Stelle besonders für die Überladung an: Der Konstruktor. Schließlich kann es ja sein, daß wir Werte bereits beim Erzeugen der Klasse zuweisen und uns somit mehrere Zuweisungen sparen wollten, die wir sonst mit Hilfe der Setter-Methoden durchführen müßten.

Sinnvoll erscheint es, alle oder einige Werte bereits beim Erzeugen der Klasse zuweisen zu lassen:

Listing 3.7
Mehrfach überladene Konstruktoren

```
public class SimpleCar {

   private String manufacturer = "Unknown";
   private String color = "White";
   private double miles = 0;

   private final double KMTOMILE = 0.62137;
   private final double MILETOKM = 1.60934;

   public SimpleCar() {
      this.setManufacturer("Porsche");
      this.setColor("Blue");
   }

   public SimpleCar(String manufacturer) {
      this();
      this.setManufacturer(manufacturer);
   }

   public SimpleCar(double miles) {
      this();
      this.setMiles(miles);
   }
```

Listing 3.7 *(Forts.)* Mehrfach überladene Konstruktoren

```
    public SimpleCar(String manufacturer, double miles) {
       this(manufacturer);
       this.setMiles(miles);
    }

    public SimpleCar(String manufacturer, String color) {
       this(manufacturer);
       this.setColor(color);
    }

    public SimpleCar(String manufacturer, String color,
       double miles) {
       this(manufacturer, color);
       this.setMiles(miles);
    }

    ...
}
```

Insgesamt verfügt die `SimpleCar`-Klasse nun über fünf Konstruktoren, die die Übergabe unterschiedlicher Parameter-Kombinationen zulassen:

- Standard-Konstruktor ohne Parameter
- Konstruktor, der die Anzahl der bereits gefahrenen Meilen entgegen nimmt (double)
- Konstruktor, der Hersteller und Meilen entgegen nimmt (String, double)
- Konstruktor, der Hersteller und Farbe entgegen nimmt (String, String)
- Konstruktor, der Hersteller, Farbe und Meilen entgegen nimmt (String, String, double)

Wenn wir nunmehr eine neue Instanz der `SimpleCar`-Klasse erzeugen, können wir nunmehr dies gleich unter Angabe der uns zur Verfügung stehenden Informationen machen. Sehen wir uns an, wie dies am Beispiel der `SimpleCarInvoker`-Klasse gelöst werden kann:

Listing 3.8 Aufruf eines überladenen Konstruktors

```
public class SimpleCarInvoker
{

   public static void main(String[] args) {
      SimpleCar car = new SimpleCar("BMW", "Blue", 1000);

      System.out.println(String.format(
         "You are driving a %s with %s color which " +
         "has the following status: %s",
             car.getManufacturer(),
             car.getColor(),
             car.carStatus()));
   }
}
```

In unserem Beispiel haben wir den Konstruktor mit drei Parametern verwendet und die Werte für Automarke, Farbe und Meilenstand gesetzt.

Achtung

Beachten Sie, dass Sie sich an die von der Klasse vorgegebene Reihenfolge der Parameter halten müssen – es ist für Compiler und Interpreter nicht ersichtlich, ob die erste Zeichenkette eine Automarke, eine Farbe oder eine Email-Adresse ist.

3.8.1 Konstruktor-Kaskadierung

Sehen wir uns noch einmal einen der überladenen Konstruktoren genauer an:

```
public SimpleCar(double miles) {
   this();
   this.setMiles(miles);
}
```

Listing 3.9
Konstrukor-Kaskadierung

Die Anweisung `this();` in der ersten Zeile der Methode ist interessant: Sie ruft direkt den Standard-Konstruktor der Klasse auf:

```
public SimpleCar() {
   this.setManufacturer("Porsche");
   this.setColor("Blue");
}
```

Listing 3.10
Standard-Konstruktor

Ändern wir nun in der Klasse `SimpleCarInvoker` so ab, das statt Marke, Farbe und Meilen nur die Meilen-Anzahl beim Instanzieren einer neuen `SimpleCar`-Klasse übergeben werden:

```
public class SimpleCarInvoker
{

   public static void main(String[] args) {
      SimpleCar car = new SimpleCar(10000);

      System.out.println(String.format(
         "You are driving a %s with %s color which " +
         "has the following status: %s",
             car.getManufacturer(),
             car.getColor(),
             car.carStatus()));
   }
}
```

Listing 3.11
Übergabe von nur einem Parameter beim Instanzieren

Wenn Sie das Beispiel kompilieren und ausführen, werden Sie diese Ausgabe erhalten:

You are driving a Porsche with Blue color which has the following status: nearly new

Wir besitzen also nun einen blauen Porsche mit 10.000 Meilen Gesamtfahrleistung. Sie sehen, das Konstruktor-Kaskadierung sehr sinnvoll sein kann, um bestimmte Standard-Werte in jedem Fall zu setzen. Dieses Ergebnis ist zwar vergleichbar mit dem Zuweisen von Standard-Werten zu den internen Variablen, dennoch aber um einiges flexibler, wie Sie im Verlauf dieses Kapitels noch feststellen werden.

3.8.2 Variable Argumente

Variable Argumente sind eine Neuerung, die bei Java 5 eingeführt worden ist. Eine derartige Funktionalität existierte nicht direkt in vorgehenden Versionen.

Eine Methode mit variablen Argumenten (zu der übrigens auch ein Konstruktor zählen kann), hat folgenden syntaktischen Aufbau:

```
<Rückgabetyp> <Name>(<Parametertyp> ... <Listenname>) { ... }
```

Wesentlich sind die drei Punkte zwischen dem Parametertyp und dem Namen der Liste, die die zugewiesenen Parameter aufnehmen soll.

Daraus, das hier von Liste – oder genauer von Array – die Rede ist, ergibt sich schon, dass die übergebenen Parameter durchlaufen werden müssen, da sie nicht als einzelne Variablen, sondern als Elemente der variablen Argumentsliste vorliegen.

Sehen wir uns an, wie dies in der Praxis umgesetzt werden kann. Nehmen wir dazu an, wir wollten eine Methode `define()` innerhalb der `SimpleCar`-Klasse implementieren:

Listing 3.12 Methode mit variablen Argumenten

```
/**
 * Sets some or all values for the class using variable arguments
 * @param args Arguments. Note: First String-argument is threatet
 * as Manufacturer, second one as Color. Every numeric argument is
 * treatet as mileage
 */
public void define(Object ... args) {
   boolean didSetManufacturer = false;

   for(int i=0; i<args.length; i++) {
      if(args[i] instanceof String) {
         String value = (String)args[i];
         if(!didSetManufacturer) {
            if(null != value && value.length() > 0) {
               this.setManufacturer(value);
            }
            didSetManufacturer = true;
         } else {
            if(null != value && value.length() > 0) {
               this.setColor(value);
            }
         }
      } else if(args[i] instanceof Number) {
         Number num = (Number)args[i];
         this.setMiles(num.doubleValue());
      }
   }
}
```

Gleich der Kopf der Methode zeigt an, dass wir eine variable Argumentliste verwenden werden. Der Nutzer kann hier keinen, einen oder beliebig viele Parameter übergeben. Im Methoden-Rumpf durchlaufen wir das Argument-Array.

Bei jedem Durchlauf wird überprüft, ob das aktuell zu behandelnde Argument vom Typ `String` oder `Number` ist – in ersterem Fall wird davon ausgegangen, das zunächst ein Hersteller gesetzt und danach die Farbe bestimmt werden soll, in letzterem Fall wird die übergebene Zahl als Meilen-Angabe interpretiert. Dies geschieht mit allen übergebenen Parametern.

Wenn wir nun die Methode innerhalb der `SimpleCarInvoker`-Klasse ansprechen wollten, könnten wir folgenden Code verwenden:

```
SimpleCar car = new SimpleCar();
car.define("Chrysler", "Yellow", 26483);
```

Listing 3.13
Aufruf der Methode define mit einer variablen Argumentliste

Wenn Sie das Beispiel kompilieren und ausführen, werden Sie diese Ausgabe erhalten:

You are driving a Chrysler with Yellow color which has the following status: used car

Nehmen wir nun, sie wüßten nicht, welche Farbe das durch die `SimpleCar`-Klasse definierte Auto haben soll. Dann würden Sie die Methode `define()` möglicherweise so aufrufen:

```
car.define("Chrysler", 26483);
```

Oder Sie drehen die Reihenfolge der Argumente einfach um:

```
car.define(26483, "Chrysler");
```

Die Zuweisung ist in beiden Fällen gültig und das Ergebnis in beiden Fällen gleich – Sie fahren einen blauen Chrysler, der irgendwie gebraucht wirkt:

You are driving a Chrysler with Blue color which has the following status: used car

Die Schlußfolgerung daraus muss lauten, daß Sie bei Verwendung von variablen Argumentlisten vorsichtig sein müssen. Sie sollten jeden einzelnen Parameter genau überprüfen und erst nach dieser Prüfung verwenden. Wenn Sie tatsächlich so kontrolliert vorgehen, sind Sie allerdings äußerst flexibel beim Umgang mit Parametern.

3.9 Zugriffs-Modifier

Im Sinne von objektorientierter Programmierung ist diese sogenannte *Kapselung* von Elementen ein wesentlicher Bestandteil des Programmier-Paradigmas. Durch sie wird sichergestellt, das nur die Elemente sichtbar sind, auf die explizit ein Zugriff gewünscht wird. Alle anderen Elemente werden verborgen und nur aus der betreffenden Klasse heraus angesprochen. Durch diese Trennung der Elemente in sichtbare und unsichtbare Elemente sorgen Sie für mehr Klarheit und eine bessere Wartbarkeit Ihrer Lösungen.

Zugriffs-Modifier regeln den Zugriff auf Klassen, Methoden oder Variablen. Mit ihrer Hilfe wird bestimmt, ob und falls ja, wie Elemente sichtbar oder unsichtbar für andere Elemente sind. Derartige Festlegungen bieten sich insbesondere im Hinblick auf private Variablen und Methoden von Klassen an – schließlich sollen diese vor aufrufenden oder abgeleiteten Elementen geschützt sein, so das niemand Elemente Ihrer internen Logik nutzen oder interne Variablen direkt modifizieren kann.

Zugriffs-Modifier sind Schlüsselworte, die Sie Klassen-, Methoden- oder Variablen-Deklarationen hinzufügen können. Auf Ebene der Zugriffsrechte unterscheidet Java folgende vier Modifier: Standard-Modifier, `public`, `protected` und `private`.

3.9.1 Standard-Modifier

Der Standard-Modifier wird immer dann angewendet, wenn kein anderer Modifier angegebenen worden ist. Der Standard-Modifier erlaubt einen Zugriff auf das entsprechende Element aus der gleichen Klasse und anderen Klassen des gleichen Pakets heraus.

Um einen Standard-Zugriffsmodifier zu verwenden, schreiben Sie eine Klassen-, Methoden- oder Variablen-Deklaration ohne weitere Schlüsselworte:

```
String aString;
void setSomething(Object value) { ... }
class someClass { ... }
```

3.9.2 public

Der Zugriffs-Modifier `public` gibt den Zugriff auf ein Element frei. Aus allen anderen Klassen kann nun auf das Element ohne Einschränkungen zugegriffen werden.

Sinnvoll ist der Einsatz des Zugriffsmodifiers `public` in folgenden Szenarien:

- Sie möchten eine Variable, Methode oder Klasse öffentlich zugänglich machen
- Sie möchten eine öffentliche Konstante definieren
- Sie möchten Getter- und / oder Setter-Methoden für private Member bereitstellen
- Sie möchten Methoden wiederverwendbar gestalten und von anderen Klassen aus ansprechen

Um den Zugriffsmodifier `public` einzusetzen, schreiben Sie eine Klassen-, Methoden- oder Variablendeklaration, der Sie den Zugriffsmodifier voranstellen:

```
public String aPublicString;
public final double KMTOMILE = 0.62137;
public Object getSomeValue() { ... }
public class somePublicClass { ... }
```

3.9.3 protected

Wenn Sie den Zugriff auf Elemente nur für abgeleitete Klassen und auf Klassen des gleichen Pakets zulassen wollen, empfiehlt sich der Einsatz des Zugriffsmodifiers `protected`. Dies ist insbesondere bei Business-Methoden sinnvoll, die auch abgeleiteten Klassen zur Verfügung stehen sollen, aber nicht von fremden Klassen sichtbar sein dürfen.

Um Elemente als `protected` zu deklarieren, stellen Sie einfach den Modifier der entsprechenden Deklaration voran:

```
protected String someProtectedStuff;
protected void setSomeValue(Object someValue) { ... }
protected class someProtectedClass { ... }
```

3.9.4 private

Als `private` deklarierte Klassen, Methoden, Getter, Setter oder Variablen sind von außen und von abgeleiteten Klassen nicht erreichbar. Diese Elemente sind nur vom beherbergenden Element erreichbar. Eine als `private` deklarierte Methode ist also nur von der Klasse, in der sie sich befindet, erreich- und vor allem: sichtbar. So können Sie Ihre Business-Methoden, Instanz-Variablen oder Hilfsklassen vor neugierigen Blicken schützen und sicherstellen, das nur Mitglieder der jeweiligen Klasse auf das entsprechende Element zugreifen können.

Um Elemente als `private` zu deklarieren, stellen Sie einfach den Modifier der entsprechenden Deklaration voran. Innerhalb einer einfachen Klasse könnte dies beispielsweise so aussehen:

```
public class SomeClass {

   private class SomeHelperClass {
      public String doSomething(String item) {
         // ...
      }
   }

   private String somePrivateItem = "Startvalue";

   public String getSomePrivateItem {
      return this.somePrivateItem;
   }

   public void manipulateThePrivateItem() {
      SomeHelperClass helper = new SomeHelperClass();
      this.somePrivateItem =
         helper.doSomething(this.somePrivateItem);
   }

}
```

Bei diesem Beispiel können nur die Methoden `manipulateThePrivateItem()` und `getSomePrivateItem()` von außen erreicht werden. Ein direkter Zugriff auf `somePrivateItem()` oder gar die Manipulation der Variablen ist nicht möglich. Die Hilfsklasse `SomeHelperClass` ist ebenfalls nicht von außen erreichbar.

3.10 Statische Methoden und Variablen

Auf *statische Methoden* und *statische Variablen* kann ohne Erzeugen einer Klassen-Instanz zugegriffen werden. Diese Elemente „bewegen“ sich nicht innerhalb eines bestimmten Objekts, sondern sind unabhängig vom Objekten gültig und ohne Objekt-Instanzierung einsetzbar. Sie werden in der Regel verwendet, um beispielsweise Konstanten oder Hilfsmethoden, die keine Klassen-Instanz benötigen, zu erzeugen. In dem Falle dient die Klassenzugehörigkeit meist eher als Organisationsstruktur.

Statische Methoden werden als *Klassenmethoden*, statische Variablen als *Klassenvariablen* bezeichnet. Nicht-statische Methoden und -Variablen heißen dagegen *Objekt-* oder *Instanz-Methoden* bzw. *-Variablen*.

Die Deklaration von statischen Methoden und Variablen geschieht, indem Sie das Schlüsselwort *static* voranstellen. Dies kann auch in Verbindung mit anderen Zugriffsmodifiern geschehen:

```
public static int someStaticVariable;
private static void someStaticMethod() { ... }
```

Lassen Sie uns noch einen kurzen Blick auf ein Beispiel von statischen Elementen werfen. Wir erkennen so besser, was bei deren Verwendung geschieht und wie wir uns dieses Verhalten zu Nutze machen könnten:

Listing 3.14
Mit Hilfe von statischen Membern kann die Anzahl von Klassen-Instanzen gezählt werden

```
public class SampleClass {

   private static int instanceCount = 0;
   private int localCount = 0;

   SampleClass() {
      instanceCount++;
      localCount++;
   }

   public static int getInstanceCount() {
      return instanceCount;
   }

   public int getLocalCount() {
      return localCount;
   }
}
```

Die Klasse `SampleClass` besitzt eine Klassenvariable `instanceCount`. Beim Erstellen einer neuen Klassen-Instanz wird deren Wert um eins erhöht und kann anschließend mit Hilfe der statischen Getter-Methode `getInstanceCount()` abgerufen werden.

Bei der nicht als statisch deklarierten Instanz-Variablen `localCount` ist der Wert beim Abruf immer Eins, denn ihr initialer Wert ist Null und wird durch den Default-Konstruktor erhöht. Der jeweilige Inhalt von `localCount` läßt sich mit Hilfe der Getter-Methode `getLocalCount()` auslesen.

Die Klassenvariable `instanceCount` behält im Gegensatz zu `localCount` ihren Wert über die verschiedenen Instanzen der Klasse hinweg. Bei jeder neuen Instanzierung wird dieser Wert durch den Default-Konstruktor erhöht.

Der Beweis für diese Aussage kann mit Hilfe einer Methode, die immer neue Instanzen der `SampleClass`-Klasse erzeugt, leicht erbracht werden:

```
public class Invoker
{
   public static void main(String[] args) {
      System.out.println("--- START ---");
      System.out.println(String.format(
         "Current instance count: %d",
         SampleClass.getInstanceCount()));
```

```
      for(int i=0;i<10;i++) {
         System.out.print("Creating instance... ");

         SampleClass sc = new SampleClass();
         System.out.print(String.format(
            "Current instance count: %d",
            SampleClass.getInstanceCount()));
         System.out.println(String.format(
            ". Local count: %d",
            sc.getLocalCount()));
      }

      System.out.println("--- FINISHED ---");
   }
}
```

Hier geschieht nichts weiter, als das zehn neue Klassen-Instanzen der `SampleClass`-Klasse erzeugt werden. Nach der Erzeugung wird der in `instanceCount` enthaltene Zählerstand mit Hilfe der ebenfalls statischen Getter-Methode `getInstanceCount()` ausgegeben. Im Anschluß erfolgt die Ausgabe des lokalen Zählerstandes mit Hilfe von `getLocalCount()`.

Wenn Sie die Klassen kompilieren und per

```
java Invoker
```

aufrufen, werden Sie folgende Ausgabe erhalten:

--- START ---

Current instance count: 0

Creating instance... Current instance count: 1. Local count: 1

Creating instance... Current instance count: 2. Local count: 1

Creating instance... Current instance count: 3. Local count: 1

Creating instance... Current instance count: 4. Local count: 1

Creating instance... Current instance count: 5. Local count: 1

Creating instance... Current instance count: 6. Local count: 1

Creating instance... Current instance count: 7. Local count: 1

Creating instance... Current instance count: 8. Local count: 1

Creating instance... Current instance count: 9. Local count: 1

Creating instance... Current instance count: 10. Local count: 1

--- FINISHED ---

Sie sehen, statische Variablen und Methoden eignen sich sehr gut dafür, Werte auch über Instanz-Grenzen hinweg zwischenzuspeichern. Ebenfalls sehr sinnvoll ist ihr Einsatz in Verbindung mit dem Modifier `final`, wie im Weiteren gezeugt wird.

3.11 Finale Klassen, Methoden und Variablen

Der Zugriffsmodifier `final` sagt aus, dass das betreffende Element nicht mehr verändert werden dürfen. Dabei hat er je nach Einsatzgebiet unterschiedliche Bedeutungen:

- Auf Variablen angewendet, bezeichnet er eine unveränderliche Variable
- Auf Methoden angewendet, sorgt er dafür, daß von dieser Methode keine Überschreibungen mehr vorgenommen werden können
- Im Kontext einer Klassen-Deklaration besagt er, das von der entsprechenden Klasse nicht mehr abgeleitet werden kann

Der Einsatz des `final`-Schlüsselwortes ist selbsterklärend: Die Dinge, die er beschreibt, sind endgültig. Beispiele für den Einsatz des `final`-Modifiers könnten so aussehen:

```
public final String UNCHANGEABLE = "This cannot be changed!";
public final static int ATYPE = 1;
public final void getSomeValue() { ... }
public final class FinalClass { ... }
```

Beachten Sie bitte die Schreibweise bei finalen Variablen-Deklarationen: Die Konvention besagt, daß die entsprechenden Variablen-Namen komplett in Großbuchstaben geschrieben werden.

Übrigens bietet es sich in den meisten Fällen an, finale Variablen auch als statisch zu kennzeichnen – so wie dies bei der zweiten beispielhaften Variablen-Deklaration der Fall war. Dies bietet den Vorteil, das Sie die entsprechenden Variablen auch ohne eine explizite Klassen-Instanzierung einsetzen können, was Ihnen etwas Schreib-Arbeit erspart.

3.12 Vererbung

Vererbung – oder eigentlich treffender: *Ableitung* – ist eines der wesentlichsten Elemente objektorientierter Entwicklung.

Wenn Sie eine Klasse von einer anderen Klasse ableiten, dann definieren oder spezifizieren Sie Methoden oder Verhaltensweisen der neuen Klasse, die sich von der abgeleiteten Klasse unterscheiden. Klingt abstrakt? Vielleicht hilft Ihnen dieser Ansatz weiter: Sie haben bei konsequenter Anwendung von Vererbung die Möglichkeit, den Erstellungsaufwand bei neuen Klassen soweit zu minimieren, das im Idealfall die Implementierung weniger zusätzlicher Methoden oder eines neuen Konstruktors ausreichend sind – Sie sparen sich jede Menge Schreibarbeit, da die ererbten Methoden und Eigenschaften auch bei der abgeleiteten Klasse vorhanden sind.

Wußten Sie, das Sie schon mit Vererbung gearbeitet haben? Schon in dem Moment, als Sie ihre erste Klasse erzeugt haben, nahmen Sie eine Ableitung vor, denn alle Java-Klassen erben implizit von der Basis-Klasse `Object`.

Die Ableitung einer Klasse von einer anderen Klasse erfolgt unter Verwendung des Schlüsselwortes `extends` bei der Deklaration einer Klasse:

```
[<Zugriffsmodifier>] class <Name> extends <Typ> { ... }
```

Die Subklasse erbt nun alle nicht als `private` deklarierten Methoden, Klassen und Instanz-Variablen der Super-Klasse. Ein direkter Zugriff auf die Elemente der Super-Klasse, die als `private` deklariert worden sind, ist nicht möglich.

Nehmen wir an, wir hätten eine Klasse `Animal` definiert, die ein Tier repräsentieren würde. Diese Klasse könnte etwa folgenden Aufbau haben:

Ein Wort zur Sprach-Regelung: Die Klasse, von der abgeleitet wird, bezeichnet man in der Regel als *Basis-* oder *Super-Klasse*. Wir verwenden hier beide Begriffe synonym. Die abgeleitete Klasse wird gerne als *Sub-Klasse* bezeichnet. Daneben sind durchaus auch die Begriffe *Ober-* und *Unter-Klassen* üblich.

```
/**
 * Repräsentiert ein Tier
 */
public class Animal
{

   private String name = "Lion";
   private String helloMessage = "Huuuuar!";

   public Animal() {}

   public Animal(String name, String helloMessage) {
      this.setName(name);
      this.setHelloMessage(helloMessage);
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }

   public void setHelloMessage(String helloMessage) {
      this.helloMessage = helloMessage;
   }

   public String getHelloMessage() {
      return helloMessage;
   }
}
```

Listing 3.15
Ein beliebiges Tier wird durch die Klasse Animal repräsentiert

Der Aufbau der Klasse ist bewußt recht einfach gehalten, schließlich handelt es sich bei der `Animal`-Klasse um eine Basis-Klasse, von der eine Ableitung erst vorgenommen werden soll. Also entscheiden wir uns für den kleinsten gemeinsamen Nenner aller möglichen Ableitungen und implementieren eine Klasse, die nur den Namen des Tiers und dessen Begrüßung für Fremde verwaltet.

Ableitung sind bei dieser Basis-Klasse nicht nur denkbar, sondern sogar nötig:

```
/**
 * Ein Hund
 */
public class Dog extends Animal
{
   public Dog() {
      this.setHelloMessage("Wuff!");
      this.setName("dog");
   }
}
```

Listing 3.16
Ableitung einer Klasse von der Basis-Klasse SimpleCar

Die Ableitung erfolgt mit Hilfe des `extends`-Schlüsselwortes, das angibt, von welcher Klasse geerbt werden soll:

```
<Zugriffsmodifier> class <Name> extends <Super-Klasse>
```

Achtung

Java beherrscht keine *Mehrfachvererbung*! Dies bedeutet nicht, dass von einer Super-Klasse nicht mehrfach abgeleitet werden kann, sondern, dass eine Sub-Klasse nicht von mehr als einer Super-Klasse erben darf. Dies ist ein echter Vorteil gegenüber anderen Programmiersprachen, denn somit entfällt ein Konstrukt, das praktisch nur zu Komplikationen geführt hat. Mehrfach-Vererbung wird bei Java in Form von *Interfaces* nachempfunden. Hierzu später mehr.

Die von `Animal` abgeleitete `Dog`-Klasse verfügt neben den ererbten Methoden nur über einen eigenen Konstruktor. Dieser setzt Namen und Willkommens-Begrüßung, so daß die Klasseninstanz nur erzeugt werden muss und dann ohne weitere Arbeit verwendet werden kann. Alle übrigen Methoden bleiben unangetastet.

3.12.1 Polymorphie im Einsatz

Sehen wir uns nun kurz noch an, wie eine neue Instanz der Klasse erzeugt werden kann:

Listing 3.17 Instanzierung der abgeleiteten Klasse

```
/**
 * Erzeugen einer neuen Animal-Instanz
 */
public class AnimalInvoker
{
   public static void main(String[] args) {
      Animal animal = new Dog();

      System.out.println(String.format(
         "You own a %s which welcomes strangers with "
         + "a loud and friendly %s",
         animal.getName(), animal.getHelloMessage()));
   }
}
```

Die Instanzierung der Klasse erfolgt wie gewohnt, mit dem einzigen Unterschied, das wir einer Variablen vom Typ `Animal` eine Instanz der von `Animal` abgeleiteten Klasse `Dog` zuweisen – womit wir uns eines der Wesensmerkmale der Polymorphie zu nutze machen, um eine Sub-Klasse nutzen zu können, denn diese behält alle Methoden und Argumente der Super-Klasse.

Der genaue Typ der eingesetzten Klasse ist uns zum Zeitpunkt des Schreibens der Klasse nicht entscheidend, denn so sind wir in der Lage, den Code wiederverwendbarer zu halten.

Sollten wir eine explizite Instanz der abgeleiteten Klasse benötigen, könnten wir nach einer kurzen Prüfung immer noch in den entsprechenden Typ konvertieren:

```
if(animal instanceof Dog) {
   Dog dog = (Dog)animal;
}
```

Der Einsatz von polymorphen Mechanismen und die Zuweisung von Sub-Klassen zu Variablen vom Typ der Super-Klasse bewahrt uns davor, uns ständig den tatsächlichen Typ der Sub-Klasse kennen zu müssen. In Form des *Factory-Patterns* werden wir davon noch viel intensiver Gebrauch machen.

Wenn Sie die Klassen kompilieren und von der Kommandozeile aufrufen, sollten Sie über Ihr neues Haustier und dessen Willkommens-Gruß informiert werden:

You own a dog which welcomes strangers with a loud and friendly Wuff!

Verwenden wir nun statt einer `Dog`- eine `Cat`-Klasse:

Listing 3.18
Auch eine Cat-Klasse kann von der Animal-Klasse abgleitet sein

```
/**
 * Eine Katze
 */
public class Cat extends Animal
{
   private String catName = "Minka";

   public Cat() {
      this.setHelloMessage("Miau!");
      this.setName("cat");
   }

   public String getCatName() {
      return this.catName;
   }

   public void setCatName(String catNameToSet) {
      if(null != catNameToSet && catNameToSet.length() > 0) {
         this.catName = catNameToSet;
      }
   }

   public String getName() {
      return super.getName() + " called "
         + this.getCatName();
   }
}
```

Die `Cat`-Klasse führt eine neue Eigenschaft ein, die es erlaubt, den Rufnamen des Tieres zu setzen. Realisiert wird diese Eigenschaft über die Instanz-Variable `catName`. Der Zugriff findet mit Hilfe der Getter- und Setter-Methoden `getCatName()` und `setCatName()` statt.

Um den Namen des Tieres auch in die Ausgabe der `AnimalInvoker`-Klasse einzufügen, ohne deren Code ändern zu müssen, `überschreiben` wir stattdessen die von der Super-Klasse geerbte Methode `getName()`. Beachten Sie dabei auch, wie auf die Methode in der Super-Klasse zugegriffen wird:

```
return super.getName() + " called " + this.getCatName();
```

Der Zugriff geschieht mit Hilfe des Schlüsselworts `super`, das immer auf die Super-Klasse verweist. Wir setzen also die neue Rückgabe aus der alten Rückgabe und dem Namen unserer Katze zusammen.

Jetzt ändern wir den Code der `AnimalInvoker`-Klasse noch so ab, daß statt eines `Dog`- nunmehr ein `Cat`-Objekt instanziert wird:

Listing 3.19 Die geänderte AnimalInvoker-Klasse instanziert nunmehr ein Cat-Objekt

```
public class AnimalInvoker
{
   public static void main(String[] args) {
      Animal animal = new Cat();

      System.out.println(String.format(
         "You own a %s which welcomes strangers with "
         + "a loud and friendly %s",
         animal.getName(), animal.getHelloMessage()));
   }
}
```

Die Ausgabe bestätigt es: Aus dem namenlosen Hund wurde Minka, eine Katze:

You own a cat called Minka which welcomes strangers with a loud and friendly Miau!

Die für uns angenehmen Vorteile der Ableitung und des Einsatzes von Polymorphie sind also:

- Die Ersparnis von Schreibarbeit
- Die Wiederverwendbarkeit bereits geschriebenen Codes
- Die Rückwärts-Kompatibilität von abgeleiteten Klassen zur Super-Klasse
- Die Möglichkeit, mit Hilfe von `super` auf die erreichbaren Methoden, Eigenschaften, Variablen und Implementierungen der Super-Klasse zugreifen zu können

Leider ist der hier gezeigte Ansatz der Erzeugung Instanzen von Sub-Klassen von Animal nicht optimal, denn jedes Mal, wenn eine andere Sub-Klassen-Instanz erzeugt werden soll, muss AnimalInvoker geändert werden.

Wir benötigen daher eine Methode, die uns entweder eine `Animal`-, `Cat`- oder `Dog`-Instanz zurückgibt. An dieser Stelle kommt das *Factory-Entwurfsmuster* (*Factory-Pattern*) ins Spiel.

3.12.2 Factory-Prinzip

Stellen Sie sich die Funktionsweise einer *Factory* genau so vor, wie Sie sich eine Fabrik aus Sicht eines Unternehmers vorstellen würden: Die Fabrik kann grundsätzlich alles bauen – man muss ihr nur sagen, was benötigt wird. Das in der Realität hier noch einige unwesentliche Faktoren wie etwa Geld, Arbeiter, Qualifizierung, Rohstoff-Situation oder Ähnliches eine Rolle spielen könnte, ignorieren wir einfach, ebenso wie den Fakt, das man Tiere nicht einfach bauen kann.

An die Fabrik ergeht nunmehr der Auftrag, ein bestimmtes Tier zu bauen, und sie tut es. Wie genau dies vor sich geht, interessiert dabei die aufrufende Methode nicht. Die Fabrik gibt am Ende das fertige Tier zurück – und dieses kann nun dressiert, abgerichtet, oder liebkost werden.

Wenn wir ein derartiges *Factory-Pattern* umsetzen wollten, könnte dies für unser Beispiel etwa so aussehen:

Listing 3.20
Factory für das Erstellen von Animal-Instanzen oder abgeleiteten Klassen

```
/**
 * Factory für die Erstellung verschiedener Animal-Instanzen
 */
public class AnimalFactory
{
   public static final int UNKNOWN = 0;
   public static final int DOG = 1;
   public static final int CAT = 2;
   public static final int COW = 3;
   public static final int DUCK = 4;

   public static Animal createAnimal() {
      return createAnimal(UNKNOWN);
   }

   /**
    * Gibt eine Animal-Instanz oder eine Sub-Klasse zurück
    * @param type Typ des Tieres
    */
   public static Animal createAnimal(int type) {
      Animal result = null;

      switch(type) {
         case AnimalFactory.DOG:
            result = new Dog();
            break;
         case AnimalFactory.CAT:
            result = new Cat();
            ((Cat)result).setCatName("Sylvester");
            break;
         case AnimalFactory.COW:
            result = new Animal();
            result.setName("Cow");
            result.setHelloMessage("Muuuuuh!");
            break;
         case AnimalFactory.DUCK:
            result = new Animal();
            result.setName("Duck");
            result.setHelloMessage("Naknaknak");
            break;
         default:
            result = new Animal();
      }

      return result;
   }
}
```

Die `AnimalFactory` hat einen recht einfachen Aufbau bekommen – jedenfalls für eine Klasse, die das *Factory-Pattern* implementiert: Sie definiert zunächst einige Konstanten, die bestimmte Tierarten repräsentieren. Außerdem stellt sie die überladene Methode `createAnimal()` zur Verfügung. Bei Aufruf der Methode ohne Parameter wird die überladene Version der Methode mit dem Wert der konstanten Variablen `UNKNOWN`, der für ein unbekanntes Tier steht, aufgerufen.

Bei direktem Aufruf der Methode `createAnimal()` unter Verwendung eines ganzzahligen Wertes, der den Wert einer der definierten Konstanten repräsentiert, wird eine neue *Animal*-Instanz, die mit den für das entsprechende Tier vorgesehenen Werten belegt ist, zurückgegeben.

Sollte ein dem Wert der konstanten Variablen `DOG` oder `CAT` entsprechender Wert übergeben worden sein, erfolgt die Rückgabe einer entsprechend initalisierten `Dog`- oder `Cat`-Instanz.

Verwenden der Factory

Innerhalb einer aufrufenden Klasse gestaltet sich der Einsatz der `AnimalFactory` zum Erzeugen von Klassen-Instanzen fast noch einfacher als das Erzeugen von neuen Instanzen:

Listing 3.21
Erstellen einer Instanz der Animal-Klasse oder abgeleiteter Klassen per Factory

```
public class AnimalInvoker
{
   public static void main(String[] args) {
      int animalType =
         (int)(Math.random() * (AnimalFactory.DUCK + 1));
      Animal animal = AnimalFactory.createAnimal(animalType);

      System.out.println(String.format(
         "You own a %s which welcomes strangers with "
         + "a loud and friendly %s",
         animal.getName(), animal.getHelloMessage()));
   }
}
```

Am Anfang der Methode main() wird mit Hilfe von `Math.random` eine Zufallszahl erzeugt, deren Wert zwischen Null und einschließlich Vier liegt. Dies entspricht dem Wertebereich, der innerhalb der `AnimalFactory`-Klasse definiert worden ist.

Die erzeugte Zufallszahl wird als Typ des von der `AnimalFactory` zu erzeugenden Instanz-Objekts angenommen und deren Methode `createAnimal()` als Parameter übergeben. Die von der Klasse zurückgegebene `Animal`-Instanz wird dann für die gewohnte Ausgabe verwendet.

Da der Typ des Tieres zufällig bestimmt worden ist, können wir an dieser Stelle natürlich nicht voraussagen, welche Ausgabe bei einem Aufruf der Klasse erzeugt wird. Wiederholen Sie den Vorgang einfach mehrmals – Sie werden feststellen, dass die Ausgabe differiert, ohne daß wir den Quellcode unserer Applikation ändern mussten.

3.13 Abstrakte Klassen und Methoden

Mit Hilfe von *abstrakten Klassen* definieren Sie Basis-Funktionalitäten, die von ableitenden Klassen überschrieben werden können oder sollen.

Achtung

Abstrakte Klassen können nie direkt instanziert werden. Sie können aber beispielsweise als Rückgabe einer Methode – etwa in Form einer *Factory* – auftreten. Ein Konstrukt wie etwa

```
abstract class AbstractClass {
   // ...
}
AbstractClass myClass = new AbstractClass();
```

kann es bei abstrakten Klassen nicht geben und würde auch folgerichtig beim Kompilieren eine Fehlermeldung provozieren.

Wenn Sie auf abstrakte Klasse oder Methoden stoßen oder selbst ein derartiges Element definieren wollen, dann begreifen Sie das Element als eine Art Platzhalter für von ihm abgeleitete Elemente, auch wenn Sie diese zu diesem Zeitpunkt nicht kennen und auch nicht kennen müssen.

Ein abstraktes Element wird immer durch den Zugriffsmodifier `abstract` definiert. Mögliche Einsatzbereiche sind auf Klassen- und Methoden-Ebene zu suchen:

```
public abstract class SomeClass { ... }
abstract void doSomething() { ... }
```

Der Einsatzbereich von abstrakten Klassen und -Methoden ist klar umrissen: Diese Elemente definieren Funktionalitäten und Verhaltensweisen, deren genaue Implementation den ableitenden Klassen überlassen bleibt oder deren Grundfunktionalität von ableitenden Klassen verwendet werden kann. Abstrakte Klassen stellen also meist auch Implementationen bereit, auf die später zurückgegriffen werden kann.

3.13.1 Deklaration einer abstrakten Klasse

Sehen wir uns nun an, wie wir eine abstrakte Klasse deklarieren können:

```
public abstract class AbstractClass {

   public int increment(int value) {
      return ++value;
   }
}
```

Listing 3.22
Abstrakte Klasse, die bereits Basis-Funktionalitäten bereit stellt

Die abstrakte Klasse `AbstractClass` verfügt über eine bereits implementierte Methode namens `increment()`. Diese Methode erhöht den Wert der übergebenen Zahl und gibt das Ergebnis zurück. Durch die bereits implementierte

Methode ist diese Klasse nicht nur als Mutter aller abgeleiteten Klassen zu verstehen, sondern stellt ebenfalls eine Basis-Funktionalität bereit, auf die von Ableitungen zurückgegriffen werden kann.

3.13.2 Implementieren der abstrakten Klasse

Da eine abstrakte Klasse nicht direkt instanziert werden kann, wird sie mehr in Form eines Platzhalters für ihre Ableitungen zum Einsatz kommen. Die einfachste mögliche Form der Implementierung einer Ableitung der `AbstractClass`-Klasse könnte dann so aussehen:

Listing 3.23
Eine simple Implementierung der abstrakten Klasse

```
public class AbstractClassImpl extends AbstractClass { }
```

Die Ableitung von der Super-Klasse geschieht exakt so, wie es auch bei einer nicht abstrakten Klasse der Fall wäre: Mit Hilfe des Schlüsselwortes `extends`.

Wissend, das in der `AbstractClass`-Klasse bereits Basis-Funktionalitäten existieren, können wir nun recht einfach mit einer konkreten Implementierung arbeiten:

Listing 3.24
Einbinden einer Implementierung der AbstractClass-Klasse

```
public class AbstractClassInvoker {

   public static void main(String args[]) {
      AbstractClass impl = new AbstractClassImpl();
      int value = (int)(Math.random() * 1000);

      System.out.println("Incrementing value of: " + value);
      System.out.println("Result is: " + impl.increment(value));
   }
}
```

In unserem Code ist es uns egal, wie genau die Implementierung arbeitet – wir verwenden eisern den Typ der Basis- bzw. abstrakten Klasse. Tatsächlich ist der Variablen vom Typ `AbstractClass` natürlich eine Instanz der `AbstractClassImpl`-Klasse zugewiesen – und die von der Super-Klasse geerbte Methode `increment()` nimmt die zuvor generierte Zufallszahl entgegen und erhöht sie.

Die Ausgabe wird bei jedem Aufruf eine andere sein, könnte aber mit ein wenig Glück (oder wenn Sie es lange genug versuchen) auch so aussehen:

Incrementing value of: 820

Result is: 821

Überschreiben und Implementieren von Methoden einer abstrakten Klasse

Natürlich werden Sie deutlich häufiger die von der abstrakten Super-Klasse bereitgestellten oder definierten Methoden überschreiben beziehungsweise implementieren.

Dies könnte in diesem Beispiel beispielsweise so aussehen:

Listing 3.25
Die Klasse DoubleIncrementor überschreibt die Methode increment der Basis-Klasse

```
public class DoubleIncrementor extends AbstractClass {

   public int increment(int value) {
      int result =super.increment(value);
      return ++result;
   }

   public static void main(String args[]) {
      AbstractClass impl = new DoubleIncrementor();
      int value = (int)(Math.random() * 1000);

      System.out.println("Incrementing value of: " + value);
      System.out.println("Result is: " + impl.increment(value));
   }
}
```

Auch hier erfolgt die Ableitung von der Super-Klasse mit Hilfe des `extends`-Schlüsselwortes.

Diesmal implementiert die Klasse eine eigene `increment()`-Version, in der mit Hilfe des `super`-Schlüsselworts auf die nunmehr von außen nicht mehr erreichbare Methode `increment()` der Basis-Klasse zugegriffen wird.

Das diese Implementierung der `AbstractClass`-Klasse übergebenen Wert um Zwei erhöht, werden Sie leicht feststellen können, denn die Ausgabe könnte so aussehen:

Incrementing value of: 263

Result is: 265

3.13.3 Definition abstrakter Methoden

Abstrakte Methoden funktionieren nach dem selben Prinzip wie abstrakte Klassen: Ableitende Klassen sind gezwungen, die entsprechende Methode zu implementieren. Die Syntax zur Deklaration einer abstrakten Methode sieht dabei etwas anders aus, als Sie dies von normalen Methoden gewöhnt sind:

```
<Modifier> abstract <Rückgabetyp> <Name>(<Parameter>);
```

Achten Sie speziell darauf, dass sowohl der Zugriffsmodifier `abstract` als auch das Semikolon am Ende der Deklaration obligatorisch sind.

Achtung

Beachten Sie, das abstrakte Methoden nur innerhalb von Klassen erlaubt sind, die ebenfalls als `abstract` deklariert sind. Definieren Sie eine Methode als `abstract`, müssen Sie folglich auch die Klasse als `abstract` deklarieren.

Wenn Sie abstrakte Methoden deklarieren, können Sie für die entsprechende Methode keinen Rumpf – Sie sind also nicht in der Lage, eine Basisfunktionalität zu definieren.

Um dies zu verdeutlichen, hier ein kurzes Beispiel:

Listing 3.26
Innerhalb der Klasse wird eine abstrakte Methode deklariert

```
public abstract class AbstractMethods {

   public final int increment(int value) {
      return ++value;
   }

   public abstract int decrement(int value);
}
```

Innerhalb der abstrakten Klasse `AbstractMethods` deklarieren wir neben der bereits fertig umgesetzten Methode `increment()` (die wir hier übrigens als `final` kennnzeichnen, damit sie von ableitenden Klassen nicht überschrieben werden kann) eine von ableitenden Klassen zu implementierende Methode `decrement()`.

Eine ableitende Klasse sollte dann diesen Aufbau haben:

Listing 3.27
Ableitung von der Abstract-Methods-Klasse mit implementierter decrement-Methode

```
/**
 * Implementierung der AbstractMethods-Klasse
 */
public class AbstractMethodsImpl extends AbstractMethods {

   public int decrement(int value) {
      return --value;
   }
}
```

Die Ableitung von der Super-Klasse findet unter Verwendung des `extends`-Schlüsselworts statt. Die eigentliche Implementierung der `decrement()`-Methode ist nicht wirklich umfangreich ausgefallen, verringert sie doch lediglich den übergebenen Wert, bevor sie ihn zurückgibt, und verzichtet dabei auch noch auf sämtliche Prüfungen.

Zuletzt benötigen wir noch eine Klasse mit einer statischen `main()`-Methode, die von der Kommandozeile aus erreichbar ist:

Listing 3.28
Die AbstractMethods-Invoker-Klasse arbeitet mit der Implementierung der AbstractMethods-Klasse

```
public class AbstractMethodsInvoker {

   public static void main(String args[]) {
      AbstractMethods impl = new AbstractMethodsImpl();
      int value = (int)(Math.random() * 1000);

      System.out.println("Incrementing value of: " + value);
      System.out.println("Result is: " + impl.increment(value));
      System.out.println("Result of decrement is: " +
         impl.decrement(value));
   }
}
```

Nach der Erzeugung einer Zufallszahl im Bereich von 0 bis 1000 wird diese durch die Methoden `increment()` und `decrement()` erhöht und höchstwahrscheinlich auch verringert. Genau läßt sich dies leider nicht im Voraus sagen, da die tatsächliche Implementierung einer abstrakten Methode der ableitenden Klasse überlassen bleibt und von der Super-Klasse nicht zu beeinflussen ist.

Wenn Sie das Beispiel kompilieren und es aufrufen, werden Sie eine Ausgabe erhalten, die ähnlich dieser sein dürfte:

Incrementing value of: 828

Result is: 829

Result of decrement is: 827

3.14 Interfaces

Schnittstellen oder *Interfaces* (beide Begriffe sind synonym zu verwenden) definieren ebenso wie abstrakte Elemente Verhaltensweisen und Methoden, implementieren sie aber im Gegensatz zu diesen niemals selbst. Eine Klasse kann zwar nur von genau einer anderen Klasse erben, dafür aber beliebig viele Interfaces implementieren.

Interfaces sind, da sie reine Deklarationen darstellen, noch abstrakter als abstrakte Klassen – sie stellen quasi Baupläne dar, und lassen dem Baumeister freie Hand, wie er sie umsetzt. Die Verwendung von Interfaces erfordert also – aufgrund der rein deklarativen Beschaffenheit – mehr Arbeit vom Entwickler, sorgt aber gleichzeitig für deutlich mehr Flexibilität.

Verwenden Sie Interfaces immer dann, wenn Sie angeben wollen, welche Methoden oder statische Variablen von implementierenden Klassen bereitgestellt werden sollen und Sie davon ausgehen, das eine derartige Definition zusammen mit anderen Definitionen eingesetzt werden wird oder wenn Sie eine bestimmte Mindest-Struktur vorgeben wollen, ohne daß Sie sich für die konkrete Implementierung interessieren.

Setzen Sie dagegen auf abstrakte Klassen, wenn Sie eine Implementierung vorgeben wollen, die von abgeleiteten Klassen überschrieben werden kann, aber nicht muss. Sie können so Basis-Funktionalitäten definieren, auf die ableitende Klasse zurückgreifen kann.

3.14.1 Ein Interface definieren

Die Definition eines Interfaces hat syntaktisch große Ähnlichkeit mit einer Klassendefinition:

```
interface <Name> { ... }
```

Niemand schreibt vor, daß ein Interface tatsächlich auch Variablen und Methoden beinhalten muss. Das einfachste mögliche Interface könnte also so aussehen:

```
interface Nameable {}
```

Tatsächlich gibt es Interfaces, die einfach nur dafür gedacht sind, um bestimmte Eigenschaften zu dokumentieren – mit Hilfe des `instanceof`-Vergleichsoperators könnte das Vorhandensein der entsprechenden Eigenschaft ermittelt werden – und deshalb einen leeren Rumpf haben.

In der Praxis werden Sie aber in der überwiegenden Anzahl der Fälle innerhalb eines Interfaces Methoden und / oder Variablen deklarieren. Dabei sollten Sie folgende zwei Einschränkungen berücksichtigen:

- Methoden innerhalb einer Schnittstelle sind immer als `public abstract` deklariert
- Variablen sind immer als öffentliche Konstanten ausgeführt und besitzen den Zugriffmodifier `public static final`.

Andere Zugriffsmodifier – etwa `protected` oder `private` – können innerhalb von Schnittstellen nicht verwendet werden.

Umgesetzt in einem Interface könnte sich folgender Code ergeben:

Listing 3.29 Interface Nameable

```
interface Nameable {

   public abstract String getName();

   void setName(String name);
}
```

Unser Interface `Nameable` definiert, das ableitende Klassen zwei Methoden implementieren müssen: `getName()` und `setName()`. Dieses Getter-/Setter-Paar soll uns den Zugriff auf den Namen des jeweiligen Objekts erlauben – ohne das uns interessieren muss, ob und wo diese Information tatsächlich gehalten wird.

Beide Methodensignaturen sind hinsichtlich ihrer Zugriffsmodifier identisch – bei `getName()` wird explizit angegeben, das die Methode abstrakt und öffentlich ist, wogegen `setName()` diese Information implizit mitführt. Um dieses Verhalten müssen Sie sich nicht kümmern – das erledigt Java für Sie.

3.14.2 Implementieren eines Interfaces

Anders als bei der Ableitung von Klassen können Sie mehrere Interfaces in einem Objekt implementieren und somit eine einfache Form der Mehrfach-Vererbung simulieren. Zu diesem Zweck ergänzen Sie dessen Deklaration um das Schlüsselwort `implements`:

```
[<Zugriffsmodifier>] class <Name> [extends <Klasse>] implements
<Interfaces> { ... }
```

Achtung

Das Erben von einer Super-Klasse und das Implementieren von Interfaces geschieht oftmals parallel. Beachten Sie deshalb, das die Angabe, welche Interfaces implementiert werden, stets nach dem `extends`-Schlüsselwort für das Erweitern einer Klasse kommt.

Innerhalb der so definierten Klasse müssen nun die vom Interface erwarteten Methoden implementiert werden. Übertragen auf das Beispiel-Interface `Nameable` könnte dies dann so aussehen:

Listing 3.30 Implementierung des Nameable-Interfaces

```
public class NameableImpl implements Nameable {

   private String name;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

Im Kopf der Klasse erfolgt mit Hilfe des `implements`-Schlüsselwortes die Angabe, welche Interfaces implementiert werden sollen. In unserem Fall ist dies nur `Nameable` – mehrere Implementierungen trennen Sie einfach durch ein Komma zwischen den Namen der Interfaces.

Die Methoden `getName()` und `setName()` fungieren in unserer Implementierung als Getter und Setter für eine lokale Variable `name`.

Um nun so generisch wie möglich auf Methoden und Variablen einer Interface-Implementierung zugreifen zu können, empfiehlt es sich, so weit als möglich mit dem Schnittstellen-Typ zu arbeiten:

```
Nameable impl = new NameableImpl();
impl.setName("Paula");
```

Der Vorteil dieser Vorgehensweise besteht darin, dass die eigentliche Implementierungsklasse von einem beliebigen Typ sein kann, was unsere Applikation nicht interessieren muss. Wir arbeiten mit dem Typ, den wir kennen und den wir auch erwarten: *Nameable*.

Ein Wort noch zur Vererbung: Wenn die Super-Klasse, von der Sie erben lassen, ein Interface implementiert, vererbt sich dies auch auf die abgeleitete Klasse. Sie müssen die entsprechende Angabe mit Hilfe des `implements`-Schlüsselwortes also nicht wiederholen.

Die Prüfung darauf, ob ein Objekt unser Interface implementiert, erfolgt mit Hilfe des `instanceof`-Vergleichsoperators:

```
if(object instanceof Nameable) {
   Nameable impl = (Nameable)object;
   impl.setName("Paula");
}
```

3.14.3 Mehrere Interfaces implementieren

Wie bereits mehrfach erwähnt, können Sie in einer Klasse mehrere Interfaces implementieren. Damit haben Sie die Möglichkeit, die verschiedenen Anforderungen der unterschiedlichen Interfaces innerhalb einer Klasse zu implementieren, was im Sinne eines verringerten Schreib- und Entwicklungsaufwandes positiv einzuschätzen ist.

Lassen Sie uns als Beispiel ein weiteres Interface neben `Nameable` implementieren: `Comparable`. Das `Comparable`-Interface ist keine neue Erfindung des Autors, sondern Bestandteil der Java-Sprachspezifikation. Es erlaubt eine von den Elementen definierte Sortierung.

Erweitern wir also die Klasse `NameableImpl`, so das wir mehrere Instanzen sortieren können:

```
public class NameableImpl implements Nameable, Comparable {

   private String name;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
```

Listing 3.31
Erweiterung der NameableImpl-Klasse um das Interface Comparable

Listing 3.31 *(Forts.)* Erweiterung der NameableImpl-Klasse um das Interface Comparable

```
    public int compareTo(Object o) {
        if(o instanceof Nameable) {
            Nameable item = (Nameable)o;
            return this.getName().compareTo(item.getName());
        }

        return 0;
    }
}
```

Im Kopf der Klasse geben wir mit Hilfe des `implements`-Schlüsselwortes an, welche Interfaces wir implementieren wollen. Mehrere Interfaces trennen wir dabei durch Kommata. Im Klassen-Rumpf erfolgt dann wie gewohnt die Implementierung der einzelnen Methoden.

Das Interface `Comparable` erwartet, dass wir eine Methode `compareTo()` bereitstellen, die die beiden Objekt-Instanzen miteinander vergleicht. Wir machen uns an dieser Stelle die Umsetzung recht einfach und verwenden die vom `String`-Objekt bereitgestellte Implementierung. Dies gilt allerdings nur für den Fall, dass das übergebene Objekt ebenfalls das Interface `Nameable` implementiert, was wir zuvor mit Hilfe des `instanceof`-Vergleichsoperators ermitteln.

Sollte das übergebene Objekt unser Interface nicht implementieren, geben wir Null zurück. Dadurch enthalten wir uns jeder Stimme und behaupten steif und fest, die beiden Objekte wären gleich.

Die drei möglichen Rückgaben der `compareTo()`-Methoden lauten übrigens:

- 0: Die Werte der zu vergleichenden Objekte sind gleich
- <0: Der Wert des Objekts ist kleiner als der Wert des als Parameter übergebenen Objekts
- >0: Der Wert des Objekts ist größer als der Wert des als Parameter übergebenen Objekts

Ein Vergleich könnte nun beispielsweise so stattfinden:

Listing 3.32 Vergleich mit Hilfe des Comparable-Interfaces

```
import java.util.ArrayList;

public class NameableInvoker {

    private static void compare(Object a, Object b) {
        if(a instanceof Comparable) {
            Comparable c = (Comparable)a;
            System.out.println(": " + c.compareTo(b));
        } else {
            System.out.println(": This is not possible. :-(");
        }
    }

    public static void main(String args[]) {
        Nameable object = new NameableImpl();
        object.setName("Paula");

        Nameable differentObject = new NameableImpl();
        differentObject.setName("Paul");
```

Listing 3.32 *(Forts.)*
Vergleich mit Hilfe des Comparable-Interfaces

```
      System.out.print("Let's compare " + object.getName()
         + " to " + differentObject.getName());
      compare(object, differentObject);

      System.out.print("And now: Let's compare an ArrayList to "
         + object.getName());
      compare(new ArrayList(), object);
   }
}
```

Die statische Methode `main()` ist der Einsprungpunkt, wenn die Klasse von der Kommandozeile aufgerufen wird. Hier erfolgt zunächst die Instanzierung von zwei `Nameable`-Instanzen, denen jeweils ein Name zugeordnet wird. Anschließend werde beide Instanzen miteinander verglichen und danach erfolgt ein Vergleich einer `ArrayList` mit einer der beiden `Nameable`-Instanzen.

Die Vergleiche werden von der privaten Hilfsmethode `compare()` durchgeführt. Diese Methode überprüft zunächst mit Hilfe des Vergleichsoperators `instanceof`, ob das zuerst übergebene Objekt das Interface `Comparable` implementiert. Sollte dies der Fall sein, wird es in eine `Comparable`-Instanz gecastet, die dann den Vergleich vornimmt. Anderenfalls wird ein entsprechender Hinweistext ausgegeben.

Achtung

Die vom Interface `Comparable` definierte Methode `compareTo()` kann eine `ClassCastException` werfen. Dies geschieht dann, wenn die beiden Instanzen, die miteinander verglichen werden sollen, nicht vom selben Typ sind. Sie können diese Ausnahme abfangen, indem Sie folgenden Code verwenden:

```
if(a instanceof Comparable) {
   Comparable c = (Comparable)a;
   try {
      System.out.println(": " + c.compareTo(b));
   } catch (ClassCastException cce) {
      System.out.println(": Comparison not possible. :-(");
   }
}
```

Sichern Sie also den Vergleich mit Hilfe eines `try-catch`-Blocks ab.

Wenn Sie die Klassen kompilieren, sollten Sie folgende Ausgabe erhalten:

Let's compare Paula to Paul: 1

And now: Let's compare an ArrayList to Paula: This is not possible. :-(

Diese Aussagen sind nicht schwer zu interpretieren:

- Der Wert von „Paula" ist größer als der Wert von „Paul"
- Die ArrayList-Klasse implementiert das Interface Comparable nicht und kann deshalb nicht mit einer Nameable-Instanz verglichen werden

3.14.4 Interface-Ableitung

Interfaces können voneinander abgeleitet werden. Und noch viel schöner: Ein Interface kann von mehreren Interfaces gleichzeitig erben – das ist klassische *Mehrfach-Vererbung*! Der Vorteil für den Entwickler: Interfaces können kaskadierend gebaut und kleinteilig gehalten werden. So verringert sich der Schreibaufwand und das Rad muss nicht jedes Mal neu erfunden werden.

Syntaktisch gesehen funktioniert eine Interface-Vererbung ähnlich wie eine Klassen-Vererbung über das Schlüsselwort `extends`:

```
interface <Name> extends <Interfaces> { ... }
```

Wenn wir also das Interface `Nameable` erweitern und eventuell auch noch als `Comparable` markieren wollten, könnte dies so aussehen:

Listing 3.33 Das Interface AdvancedNameable erbt von Nameable und Comparable

```
public interface AdvancedNameable extends Nameable, Comparable {

   void setLastname(String lastname);
   String getLastname();
}
```

Klassen, die dieses Interface implementieren, müssen über alle fünf in den drei Schnittstellen geforderten Methoden verfügen: `getName()` und `setName()` aus `Nameable`, `compareTo()` aus `Comparable`, sowie `getLastname()` und `setLastname()` aus `AdvancedNameable`. Sollte eine der Methoden nicht implementiert sein, ist auch die Schnittstelle nicht umgesetzt.

Die Klassen, die `AdvancedNameable` implementieren, können für andere Klassen durch das Konzept der Polymorphie in verschiedene Rollen schlüpfen: Sie können von ihrem eigentlichen Typ sein, sie können vom Typ `Comparable` sein, sie sind vom Typ `Nameable` und natürlich auch vom Typ `AdvancedNameable`. Je nach Bedarf kann auf den Typ mit Hilfe des `instanceof`-Operators geprüft und anschließend in den benötigten Typ gecastet werden.

Sehen wir uns an, wie dies in einer implementierenden Klasse aussehen kann:

Listing 3.34 Die Klasse AdvancedNameableImpl implementiert gleich drei Schnittstellen

```
public class AdvancedNameableImpl implements AdvancedNameable {

   private String lastname;
   private String firstname;

   public void setLastname(String lastname) {
      this.lastname = lastname;
   }

   public String getLastname() {
      return this.lastname;
   }

   public String getName() {
      return this.firstname;
   }
```

```
    public void setName(String name) {
        this.firstname = name;
    }

    public int compareTo(Object o) {
        if(o instanceof AdvancedNameable) {
            AdvancedNameable an = (AdvancedNameable)o;
            int result =
                this.getLastname().compareTo(
                    an.getLastname());

            if(0 != result) {
                return result;
            }
        }

        if(o instanceof Nameable) {
            Nameable n = (Nameable)o;
            return this.getName().compareTo(n.getName());
        }

        throw new ClassCastException("Unable to cast instance of "
            + o.getClass().getName() + " into instance of Nameable "
            + "or AdvancedNameable!");
    }
}
```

Listing 3.34 *(Forts.)*
Die Klasse AdvancedNameableImpl implementiert gleich drei Schnittstellen

Die Implementierung der Getter und Setter-Methoden, die von `Nameable` und `AdvancedNameable` definiert worden sind, ist trivial. Sie verweisen auf die Instanzvariablen `name` und `lastname`.

Interessanter ist sicherlich die Methode `compareTo()`, die vom Interface `Comparable` definiert worden ist: Hier erfolgt zunächst eine Prüfung darauf, ob das übergebene Objekt vom Typ `AdvancedNameable` ist. Sollte dies zutreffen, erfolgt der Vergleich der Objekte anhand der Nachnamen. Sollte dieser Vergleich ergeben, dass die Objekte ungleich sind, wird dies zurückgegeben und die Vergleichsoperation ist beendet.

Sollte sich allerdings herausstellen, dass die Objekte gleich sind (weil die Nachnamen gleich sind), wird exakt genauso fortgefahren, als ob das übergebene Objekt keine `AdvancedNameable`-Instanz wäre: Es erfolgt eine Prüfung darauf, ob das übergebene Objekt die Schnittstelle `Nameable` implementiert. Wenn dem so ist, dann werden die durch `getName()` abgerufenen Werte mit Hilfe der `compareTo()`-Methode der `String`-Klasse miteinander verglichen und das Ergebnis zurückgegeben.

Wenn das übergebene Objekt allerdings weder `AdvancedNameable`, noch `Nameable` implementieren sollte, wird davon ausgegangen, dass es nicht mit der Klassen-Instanz verglichen werden kann. Entsprechend wird eine `ClassCastException` geworfen, die von der aufrufenden Methode abgefangen werden sollte.

Nach soviel Theorie wieder eine kurze praktische Demonstration:

Listing 3.35
Vergleich mehrerer Objekte mit Hilfe der compareTo-Methode

```
public class AdvancedNameableInvoker {

   public static void main(String args[]) {
      AdvancedNameable object = new AdvancedNameableImpl();
      object.setName("Paula");
      object.setLastname("Meier");

      AdvancedNameable differentObject = new AdvancedNameableImpl();
      differentObject.setName("Paul");
      differentObject.setLastname("Mueller");

      Nameable thirdObject = new NameableImpl();
      thirdObject.setName("Paula");

      System.out.println(String.format(
         "Let's compare %s %s to %s %s: %d",
            object.getName(), object.getLastname(),
            differentObject.getName(),
            differentObject.getLastname(),
            object.compareTo(differentObject)));

      System.out.println(String.format(
         "And now: Let's compare %s %s to %s: %d",
            object.getName(), object.getLastname(),
            thirdObject.getName(),
            object.compareTo(thirdObject)));
   }
}
```

Innerhalb der statischen Methode `main()` werden drei Objektinstanzen erzeugt: `object` und `differentObject` sind vom Typ `AdvancedNameableImpl` und implementieren das Interface `AdvancedNameable`. Das Objekt `thirdObject` ist dagegen vom Typ `NameableImpl` und implementiert folglich die Schnittstelle `Nameable`.

Mit Hilfe der `compareTo()`-Methode der `AdvancedNameable`-Implementierung werden nun zunächst die beiden Instancen der `AdvancedNameableImpl`-Klasse verglichen. Anschließend wird eine `AdvancedNameable`-Instanz mit dem Objekt `thirdObject` verglichen, das die Schnittstelle `Nameable` implementiert.

Lassen Sie uns an dieser Stelle zwei Voraussagen treffen:

- „Paula Meier“ wird ein Ergebnis kleiner als Null einfahren, wenn man sie mit „Paula Mueller“ vergleicht. Der Grund dafür: Wenn zwei `AdvancedNameable`-Instanzen miteinander verglichen werden, wird zunächst der per `getLastname()` erreichbare Nachname verglichen – und da im Alphabet der Buchstabe „e“ vor „u“ kommt, wird der Wert von „Paula Meier“ offensichtlich kleiner als der Wert von „Paula Mueller“ sein.
- Das Ergebnis des Vergleichs von „Paula Meier“ zu „Paula“ wird Null (also „gleich“) sein, denn „Paula“ ist eine `Nameable`-Instanz. Dadurch kann nicht anhand der Nachnamen verglichen werden und „Paula Meier“ reduziert sich für die `compareTo()`-Methode auf „Paula“. Und „Paula“ ist gleich „Paula“...

Das Ergebnis wird die getroffenen Aussagen bestätigen:

Let's compare Paula Meier to Paula Mueller: -16

And now: Let's compare Paula Meier to Paula: 0

3.15 Lokale und anonyme Klassen

Lokale Klassen sind Klassen, die innerhalb einer anderen Klasse definiert worden sind. Diese Klassen besitzen in der Regel nur eine Bedeutung für die deklarierende Klasse, können aber bei Bedarf auch von anderen Klassen angesprochen werden.

Sie könnten also in einer Klasse `WrapperClass` problemlos eine in ihr enthaltene Klasse `ContainedClass` erstellen:

```
public class WrapperClass {
   public class ContainedClass {}
}
```

Diese Vorgehensweise bietet sich insbesondere für Hilfsklassen an, die von vielen Methoden der äußeren Klasse genutzt werden sollen, aber von anderen Klassen nicht unbedingt erreichbar sein müssen.

Interessant ist die Sichtbarkeit von Variablen gelöst: Aus der lokalen Klasse heraus sind Membervariablen der umschließenden Klasse sichtbar! Dies bedeutet, das auf eine in der äußeren Klasse definierte Variable ebenso zugegriffen werden kann, wie auf eine in der lokalen Klasse definierte Variable:

Listing 3.36
Verwenden lokaler Klassen

```
public class WrapperClass {

   String name;
   int numericValue;

   WrapperClass() {
      name = "Outer class";
      numericValue = 200;
   }

   public void process() {
      ContainedClass inner = new ContainedClass();
      System.out.println(inner.getOutput());
   }

   class ContainedClass {

      private String name;

      ContainedClass() {
         name = "Inner class";
      }
```

Listing 3.36 *(Forts.)*
Verwenden lokaler Klassen

```
        private String getOutput() {
            return "My name: " + name + ", Outer class name: " +
                WrapperClass.this.name + ", Number: " +
                    numericValue;
        }

    }

    public static void main(String[] args) {
        WrapperClass outer = new WrapperClass();
        outer.process();
    }
}
```

Innerhalb der Klasse `WrapperClass` wurde eine lokale Klasse `ContainedClass` definiert. `WrapperClass` deklariert zwei Membervariablen – `name` und `numericValue`. Die lokale Klasse `ContainedClass` deklariert eine eigene Membervariable `name`, die aus Sicht der lokalen Klasse die Variable `name` der äußeren Klasse verdeckt.

Im Konstruktor der beiden Klassen werden jeweils die Standard-Werte der Variablen gesetzt. Aus der statischen Methode `main()` heraus wird eine neue `WrapperClass`-Instanz erzeugt und deren Methode `process()` aufgerufen. Innerhalb dieser Methode wird eine neue `ContainedClass`-Instanz erzeugt und die Rückgabe von deren Methode `getOutput()` ausgegeben.

In `getOutput()` wird die Rückgabe erzeugt, wobei auf drei unterschiedliche Arten auf Variablen zugegriffen wird: Zunächst erfolgt der Zugriff auf die lokale Membervariable `name` der `ContainedClass`-Instanz. Der Zugriff auf die Membervariable `name` der äußeren Klasse erfolgt mit Hilfe eines seltsam anmutenden Statements:

```
", Outer class name: " + WrapperClass.this.name
```

Dieser Ausdruck, der immer der Form

```
<Klassenname>.this
```

entspricht, erlaubt den Zugriff auf die für die lokale Klasse eigentlich nicht sichtbare Variable `name` der äußeren Klasse – diese Variable ist nicht sichtbar, weil die lokale Klasse eine eigene Membervariable namens `name` besitzt. Durch `<Klassenname>.this` können wir nun dennoch auf die Membervariable der äußeren Klasse zugreifen.

Die dritte Art des Zugriffs auf eine Variable ist der Zugriff auf eine nicht verdeckte Variable der äußeren Klasse, wie beim Abrufen des Wertes von `numericValue` demonstriert:

```
", Number: " + numericValue
```

Dieser Zugriff erfolgt ohne Qualifizierung und ohne, das per `<Klassenname>.this` explizit auf die äußere Klasse zugegriffen werden muss – der Java-Interpreter erledigt die Auflösung des Namens für uns, indem er zunächst prüft, ob in der aktuellen Methode eine derartige Variable existiert. Sollte dies nicht

der Fall sein, erfolgt eine Prüfung auf Klassen-Ebene. Wird er dort auch nicht fündig, setzt er seine Überprüfung in der äußeren Klasse fort – und zwar solange, bis er eine entsprechende Variable findet oder eine Exception geworden werden muss, weil eine derartige Variable tatsächlich nicht existiert.

Wenn Sie dieses Beispiel ausführen, werden Sie diese Ausgabe erhalten:

My name: Inner class, Outer class name: Outer class, Number: 200

3.15.1 Lokale Klassen in Methoden

Lokale Klassen müssen nicht zwingend außerhalb einer Methode oder eines Blocks definiert werden. Es ist ebenso möglich (aber nicht gebräuchlich), eine Klasse in einem Block oder einer Methode zu definieren. Dadurch haben wir die Möglichkeit, nicht nur auf Instanzvariablen der äußeren Klasse zuzugreifen, sondern ebenfalls auf die innerhalb der Methode oder des Blocks deklarierten Variablen.

Wandeln wir das gerade gezeigte Beispiel ein wenig ab:

Listing 3.37
Definition einer lokalen Klasse innerhalb einer Methode

```
public class MethodClass {

   String name;

   MethodClass() {
      name = "Outer class";
   }

   public void process() {
      final int numericValue = 400;

      class ContainedClass {
         private String name;

         ContainedClass() {
            name = "Inner class";
         }

         private String getOutput() {
            return "My name: " + name +
               ", Outer class name: " +
               MethodClass.this.name +
               ", Number: " + numericValue;
         }

      }

      ContainedClass local = new ContainedClass();
      System.out.println(local.getOutput());
   }

   public static void main(String[] args) {
      MethodClass outer = new MethodClass();
      outer.process();
   }
}
```

In diesem Beispiel wird innerhalb der Methode `process()` der äußeren Klasse `MethodClass` eine lokale Variable `numericValue` deklariert. Diese muss zwingend als `final` gekennzeichnet sein, wenn von einer inneren Klasse auf sie zugegriffen werden soll!

Anschließend erfolgt die Deklaration der inneren Klasse, die innerhalb ihrer Methode `getOutput()` auf die beiden `name`-Membervariablen von äußerer- und innerer Klassen-Instanz zugreifen. Ebenfalls erfolgt der Zugriff auf die Variable `numericValue`, die nur innerhalb der Methode existiert, in der die Klasse deklariert worden ist.

Bei Aufruf dieses Beispiels werden Sie folgende Ausgabe erhalten:

My name: Inner class, Outer class name: Outer class, Number: 400

Wie bereits erwähnt ist der Einsatz dieser Art von Klasse recht unüblich. Der Grund dafür liegt in zwei Nachteilen:

- Die Klasse ist außerhalb der aktuellen Methode oder des aktuellen Blocks nicht sichtbar und kann somit nicht weiter verwendet werden
- Der Schreibaufwand der Deklaration einer derartigen Klasse läßt sich noch verringern

Statt Klassen in Methoden zu deklarieren, setzt man oftmals das Konstrukt der anonymen Klasse ein, dem wir uns nun widmen werden.

3.15.2 Anonyme Klassen

Anonyme Klassen entsprechen weitestgehend dem Konzept lokaler Klassen innerhalb von Methoden, verringern aber den Schreibaufwand bei deren Deklaration. Dies geschieht, indem Definition und Instanzierung der Klasse in einer Anweisung erfolgen. Aus Gründen der Übersichtlichkeit werden anonyme Klassen in der Regel in Form von Ableitungen anderer Klassen oder der Implementierung von Interfaces verwendet. Ihr wichtigster Einsatzzweck ist die Definition von Event-Listenern beim Einsatz von *Swing* und *JFC*.

Lassen Sie uns dies an einem Beispiel verdeutlichen, in dem wir eine anonyme Klasse verwenden, um eine andere Klasse zu erweitern:

Listing 3.38
Deklaration und Verwendung einer anonymen Klasse

```
public abstract class AnonymousClass {

   String name;
   int numericValue;

   AnonymousClass() {
      name = "Anonymous class";
      numericValue = 200;
   }

   public abstract void process();
```

```
   public static void main(String[] args) {
      new AnonymousClass() {
         public void process() {
            System.out.println(
               "Name: " + name + ", value: " + numericValue);
         }
      }.process();
   }
}
```

Listing 3.38 *(Forts.)*
Deklaration und Verwendung einer anonymen Klasse

Die hier deklarierte Klasse `AnonymousClass` ist als `abstract` gekennzeichnet und erfodert die Implementierung der Methode `process()`, die eine wie auch immer geartete Verarbeitung vornehmen soll.

Innerhalb ihrer statischen Methode `main()` (auf die wir trotz der Kennzeichnung der Klasse als `abstract` zugreifen können), erzeugen wir eine neue Instanz der `AnonymousClass`-Klasse und implementieren dabei – wie gefordert – deren Methode `process()`. Direkt nach der Deklaration der Klasse (nach der schließenden geschweiften Klammer) rufen wir die soeben implementierte Methode `process()` auf. Dies ist möglich, da wir die Klasse ja bereits per `new` instanziert haben – wir haben hier also zwei Dinge auf einmal: Deklaration und Instanzierung.

Grundsätzlich sieht die Syntax für die Deklaration einer anonymen Klasse also so aus:

```
new <Klassenname> {
   // Implementierung
}[.<Methodenaufruf>]
```

Eine derart erzeugte anonyme Klasse kann wie eine gewöhnliche Objektinstanz an Methoden als Parameter übergeben werden. Innerhalb der Methode ist dann kein Unterschied bei der Arbeit mit der anonymen Klasse mehr feststellbar.

Darüber, ob der Einsatz anonymer Klassen im Hinblick auf Lesbarkeit oder Übersichtlichkeit Vorteile bringt, kann sicherlich gestritten werden. Ein unbestreitbarer Vorteil anonymer Klassen kann allerdings nicht wegdiskutiert werden: Sie sind absolut flexibel einsetzbar, denn sie werden dort deklariert, wo sie auch verwendet werden. Außerdem gelten für sie die gleichen Vorteile wie für lokale Klassen auch: Sie können auf Instanzvariablen der äußeren Klasse oder des umschließenden Blocks zugreifen – und dies auch, wenn sie an andere Methoden als Parameter weitergegeben worden sind.

Aus Gründen der Lesbarkeit und der Wartbarkeit sollten Sie den Einsatz anonymer Klassen jedoch auf kleine Konstrukte mit wenigen Zeilen Code beschränken.

3.16 Aufzählungen

Aufzählungen erlauben es, bestimmte Konstanten zu gruppieren und unter ihrem Namen innerhalb von Applikationen zu verwenden. In älteren Java-Versionen geschah dies, indem öffentliche Konstanten als Integer-Werte deklariert worden sind:

Listing 3.39 Herkömmliche Deklaration von Aufzählungen

```
public class OldEnums {
   public static final int DAY_SUNDAY = 0;
   public static final int DAY_MONDAY = 1;
   public static final int DAY_THUESDAY = 2;
   public static final int DAY_WEDNESDAY = 3;
   public static final int DAY_THURSDAY = 4;
   public static final int DAY_FRIDAY = 5;
   public static final int DAY_SATURDAY = 6;

   public static final int FILE_OPEN = 0;
   public static final int FILE_CLOSED = 1;
   public static final int FILE_DELETED = 2;
}
```

Zwar ist dieses Verfahren einfach und robust (wir haben es selbst in diesem Kapitel schon angewendet), es weist aber neben einigen anderen Nachteilen ein wesentliches Manko auf: Klassische Aufzählung unter Verwendung von konstanten Variablen sind nicht eindeutig und typsicher.

Der durch eine konstante Variable repräsentierte Wert kann genauso durch eine andere Variable repräsentiert werden. So können zwei konstante Variablen `FILE_OPEN` und `DAY_SUNDAY` beide den Wert `0` repräsentieren:

```
public static final int FILE_OPEN = 0;
public static final int DAY_SUNDAY = 0;
```

Innerhalb von Applikationen kann dies durchaus ein Problem darstellen, denn eine Typsicherheit und damit eine eindeutige Eingrenzung der Anwendung kann nicht garantiert werden. Nehmen wir an, wir hätten eine Klasse `Day` geschrieben, deren Konstruktor ein Element der `OldEnums.DAY`-Konstanten entgegennehmen sollte:

Listing 3.40 Die Klasse Day verfügt über einen Konstruktor, der ein Element einer Aufzählung entgegennimmt

```
public class Day {

   private int day = OldEnums.DAY_SUNDAY;

   Day(int day) {
      this.day = day;
   }
}
```

Alle nun folgenden Zuweisungen wären gültig:

```
Day day = new Day(OldEnums.DAY_SUNDAY);
Day secondDay = new Day(OldEnums.FILE_OPEN);
Day thirdDay = new Day(0);
```

Das Problem dabei ist: Ohne wirkliche Eindeutigkeit der Parameter-Typen sinkt die Sicherheit der Applikation.

3.16.1 Eindeutig und typsicher

Das Schlüsselwort *enum*, das mit Java 5 eingeführt worden ist, erlaubt es nunmehr, Aufzählungen analog zu Klassen- und Interface-Deklarationen zu definieren:

```
[Zugriffsmodifier] enum <Name> {
   Element[, ...]
}
```

Achtung

Beachten Sie, dass sich die Analogie zu Klassen und Interfaces nicht nur auf die Syntax beschränkt. Auch für alleinstehende Aufzählungen gilt: Eine Aufzählung – eine Datei gleichen Namens.

Mit Hilfe des `enum`-Schlüsselwortes und vor allem der dahinter stehenden Technologie, erschlagen wir mehrere Fliegen mit einer Klappe:

- Aufzählungen werden eindeutig und typsicher
- Die einzelnen Elemente der Aufzählung sind selbsterklärend
- Jede Aufzählung fungiert als eindeutiger Namensraum für deren Elemente – mehrere Aufzählungen können also mehrmals gleichartige Elemente definieren, und diese können dennoch unterschieden werden

Setzen wir nun zur Verdeutlichung unsere `DAY`- und `FILE`-Konstanten in Form von Aufzählungen um:

Listing 3.41
Aufzählungen unter Java 1.5

```
public class Globals {

   public enum Days {
      Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
   }

   public enum FileStatus {
      OPEN, CLOSED, DELETED
   }
}
```

Die einzelnen Elemente der Aufzählungen werden durch Kommata voneinander getrennt. Java weist ihnen selbstständig einen numerischen Wert zu, der ihrer Position in der Liste entspricht. Der erste Wert der Aufzählung hat dabei immer den Wert 0, der folgende 1 und so weiter.

Ändern wir die Klasse `Day` so ab, das sie die `Days`-Aufzählung aus der `Globals`-Klasse verwendet:

Listing 3.42
Die Day-Klasse verwendet nun Elemente der Days-Auflistung aus der Globals-Klasse

```
public class Day {

   private Globals.Days currentDay = Globals.Days.Sunday;

   Day(Globals.Days day) {
      this.currentDay = day;
   }
}
```

Werfen Sie zunächst einen Blick auf die Deklaration der internen Variable `currentDay`: Diese ist nunmehr kein `Integer`-Wert mehr, sondern ein Element vom Typ der `Globals.Days`-Auflistung. Beachten Sie auch, wie ein Standard-

Wert zugewiesen worden ist: Der Zugriff auf ein Element der Auflistung erfolgt genau so, als ob wir auf eine statische Variable zugreifen würden.

Wenn Sie nun versuchen würden, eine Element einer anderen Aufzählung oder eine Konstante zuzuweisen (auch wenn diese den gleichen Wert hätte), wird dies nicht funktionieren, da Java zunächst auf den Parameter-Typ prüft – und der muss vom Typ `Globals.Days` sein. Damit ist sichergestellt, das wirklich nur Elemente vom gewünschten Typ zugewiesen werden können, wodurch eines der größten Mankos der bisherigen Vorgehensweise beseitigt wurde.

3.16.2 Enumerations-Member durchlaufen

Mit Hilfe der neuen `enum`-Auflistungen ist es nunmehr auch möglich, die einzelnen Elemente einer Aufzählung zu durchlaufen:

Listing 3.43
Durchlaufen einer Aufzählung und Ausgabe aller Member

```
public class FileStatusItems {

   public static void main(String args[]) {
      System.out.println("Members of the FileStatus enumeration:");
      System.out.println("=====================================");
      for(Globals.FileStatus status : Globals.FileStatus.values()) {
         System.out.println(
            String.format("%d = %s",
            status.ordinal(), status));
      }
   }
}
```

Eine einfache `for`-Schleife genügt, um alle Elemente einer Auflistung auszugeben.

Der Grund dafür ist, dass uns jede Auflistung eine Methode `values()` bereitstellt. Diese liefert ein Array aus den einzelnen Aufzählungselementen, das wir anschließend durchlaufen können. Auf Ebene eines einzelnen Elements liefert uns dessen Methode `ordinal()` den repräsentierten Zahlenwert – und den Namen für das Element stellt das Element selbst dar.

Die Ausgabe der `FileStatusItems`-Klasse sollte uns über die einzelnen Elemente der `FileStatus`-Auflistung und deren Werte informieren:

Members of the FileStatus enumeration:
=====================================
0 = OPEN
1 = CLOSED
2 = DELETED

3.17 Pakete

Pakete erlauben es, Klassen, Interfaces oder Auflistungen hierarchisch zu organisieren. Somit können wir allen Elementen einen eindeutigen Namensraum voranstellen. Der praktische Nutzen erschließt sich schnell, wenn Sie Klassen entweder einfach organisieren wollen oder dazu gezwungen sind, weil Sie plötzlich gleich- oder sehr ähnlich benannte Elemente haben.

Pakete haben beim Einsatz folgende Vorteile:

- Sie organisieren Ihre Elemente in einzelne Einheiten – analog zu Ordnern auf Ihrer Festplatte oder Unterverzeichnissen in Ihrem Web-Auftritt
- Sie lösen das Benennungs-Problem – Elemente müssen nur innerhalb ihres Namensraums eindeutig benannt sein. Sie können somit je ein Element namens MyClass innerhalb eines Paketes packageA und innerhalb eines Paketes packageB definiert haben
- Pakete geben nur auf Elemente Zugriff, die als public deklariert worden sind
- Pakete erlauben es Ihnen, sprechende Organisationsnamen zu verwenden und somit mehr Klarheit über den Zweck von Elementen zu schaffen

3.17.1 Pakete definieren

Jedes Paket bildet einen eigenen Namensraum. Die Definition des Pakets, zu dem ein Element gehört, erfolgt mit Hilfe des `package`-Schlüsselworts, das vor der eigentlichen Element-Deklaration stehen muss:

```
package <Paketbezeichner>;
```

Der eigentliche Bezeichner kann dabei eine beliebige Zeichenkette sein. Die Konvention sieht dabei ein Modell vor, das am Anfang aus einem umgekehrten Domainnamen und danach einem eindeutigen Paket-Bezeichner besteht. Die einzelnen Bestandteile sind dabei analog zu Domainnamen durch Punkte getrennt. Generell sollte die Reihenfolge der Bestandteile einer Bedeutungshierarchie folgen – von Allgemein zu Speziell, von einer Länderdomain über den Namen des Entwicklers oder der Firma bis hin zum Namen des Produktes.

Ein Beispiel: Angenommen, sie besäßen die Domain *ksamaschke.de* (ohne *www* am Anfang) und wollten eine Klasse für das „MasterClass"-Buch schreiben, dann könnten Sie folgende Paket-Deklaration verwenden:

```
package de.ksamaschke.masterclass;
public class SampleClass { ... }
```

Innerhalb Ihrer Verzeichnis-Struktur müssen Sie nun den Namensraum widerspiegeln. Dabei entspricht jedes Element des Namensraums einem Verzeichnis, das unterhalb des vorhergehenden Elements liegt:

Abbildung 3.2
Verzeichnis-Struktur bei Verwendung des Namensraums de.ksamaschke.masterclass

Eine Klasse `SampleClass`, die zum Paket `de.ksamaschke.masterclass` gehört, liegt also im Verzeichnis `/de/ksamaschke/masterclass`.

3.17.2 Pakete verwenden

Um Elemente, die in Paketen organisiert sind, verwenden zu können, müssen Sie sich auf den kompletten Paketnamen beziehen und diesen der Element-Deklaration voranstellen. Wollten Sie also aus einer Applikation auf die Klasse

SampleClass aus dem Paket de.ksamaschke.masterclass zugreifen und diese verwenden, könnte das so vor sich gehen:

Listing 3.44 Verwenden einer Klasse aus einem anderen Paket

```
public class SampleClassSample {

   public static void main(String[] args) {
      de.ksamaschke.masterclass.SampleClass sample =
         new de.ksamaschke.masterclass.SampleClass();
      System.out.println(sample.sayHello());
   }
}
```

Die Syntax lautet also beim Instanzieren:

```
<Paket>.<Typ> = new <Paket>.<Typ>();
```

Wichtig ist, daß das Paket immer komplett angegeben wird. Sie können keine Platzhalter oder Kürzel definieren – wenn ein Paket zwanzig Ebenen hat, müssen Sie alle zwanzig Ebenen angeben.

3.17.3 Das import-Statement

Weil die Angabe des kompletten Palets durchaus lästig werden kann, wenn Sie dies mehrfach vornehmen müssen, verfügt Java über das import-Schlüsselwort. Dieses erlaubt es, innerhalb einer Klasse alle zu verwendenden Hierarchien verfügbar zu machen, ohne immer wieder den kompletten Namensraum der Elemente anzugeben.

Das import-Statement steht immer am Anfang einer Datei, meist sogar noch vor der Angabe des Pakets:

```
import <Paket>.<Typ>;
```

Wollten Sie beispielsweise auf die Klasse SampleClass aus dem Paket de.ksamaschke.masterclass zugreifen, könnten Sie dies mit Hilfe des import-Statement so umsetzen:

Listing 3.45 Importieren eines Pakets durch das import-Statement

```
import de.ksamaschke.masterclass.SampleClass;

public class SampleClassSample2 {

   public static void main(String[] args) {
      SampleClass sample = new SampleClass();
      System.out.println(sample.sayHello());
   }
}
```

Nach dem Importieren des Namensraums des SampleClass-Typs können wir nunmehr also statt

```
de.ksamaschke.masterclass.SampleClass sample =
   new de.ksamaschke.masterclass.SampleClass();
```

viel kürzer und einfacher

```
SampleClass sample = new SampleClass();
```

schreiben.

3.17.4 Das import-Statement mit Platzhaltern

Beim Import von Elementen spart man sich bereits in der bekannten Form eine Menge Schreibarbeit. Ebenfalls möglich ist aber die Verwendung eines Platzhalters. Dies erlaubt es, statt eines konkreten Typs eine Namensraum-Ebene anzugeben, die mitsamt der direkt untergeordneten Elemente importiert werden soll. Die Syntax ändert sich nur marginal:

```
import <Paket>.*;
```

Achtung

Erwarten Sie bei Verwendung des `import`-Statements mit dem Platzhalter-Zeichen nicht, daß auch untergeordnete Namensräume importiert werden. Diese müssen Sie weiterhin per `import`-Anweisung einbinden. Ein Beispiel soll dies verdeutlichen: Angenommen, Sie wollten Elemente der Pakete `de.ksamaschke` und `de.ksamaschke.masterclass` einbinden, dann müssen Sie dies auch entsprechend angeben:

```
import de.ksamaschke.*;
import de.ksamaschke.masterclass.*;
```

Ebenso ist es nicht möglich, den Platzhalter an einer anderen Stelle als der Angabe des Typs zu verwenden. Dieses Statement würde also nicht funktionieren:

```
import de.*.masterclass;
```

Der Nutzen des `import`-Statements kommt dann zum Tragen, wenn Sie sehr viele Typen eines Pakets einbinden und verwenden wollen. Statt beispielsweise zehn `import`-Statements für zehn Typen zu verwenden, können Sie dies – falls die Elemente im gleichen Paket liegen – per `import`-Anweisung mit Platzhalter in nur einem Statement erledigen.

Achtung

Es ist aus Performance- und Lesbarkeitsgründen immer vorzuziehen, die zu verwendenden Typen direkt zu importieren. Der Platzhalter sollte immer nur dann zum Einsatz kommen, wo eine große Anzahl (cirka ab sieben oder acht Elementen) von `import`-Statements nötig wäre, um Elemente zu importieren.

3.17.5 Namensraum-Konflikte

Mit Hilfe von Paketen können gleich benannte Elemente sehr gut voneinander getrennt werden, denn die Namen der Pakete stellen einen eindeutigen Namensraum dar. Was aber geschieht, wenn Sie zwei gleichnamige Elemente per `import`-Statement einbinden und verwenden wollen? Angenommen, Sie hätten folgenden Code:

Listing 3.46 Namensraum-Konflikt

```
import de.masterclass.java.objects.SampleClass;
import de.masterclass.java.others.*;

public class NamingConflict {

   public static void main(String[] args) {
      SampleClass sample = new SampleClass();
      SampleClass otherSample = new SampleClass();
   }
}
```

Wie verhält sich der Java-Interpreter an dieser Stelle? Ganz einfach: Er tut überhaupt nichts! Sie werden die Klasse schon nicht kompiliert bekommen, da der jeweils zu verwendende Typ nicht eindeutig identifizierbar ist.

Wenn Sie also gleichnamige Klassen aus unterschiedlichen Namensräumen importieren und verwenden wollen, müssen Sie dem Compiler schon deutlich sagen, welche Klasse Sie instanzieren wollen:

Listing 3.47 Auflösung des Namensraums-Konflikts

```
import de.masterclass.java.objects.SampleClass;
import de.masterclass.java.others.*;

public class NamingConflict {

   public static void main(String[] args) {
      de.masterclass.java.objects.SampleClass sample =
         new de.masterclass.java.objects.SampleClass();
      de.masterclass.java.others.SampleClass otherSample =
         new de.masterclass.java.others.SampleClass();
   }
}
```

Nunmehr kann das Beispiel kompiliert und ausgeführt werden.

3.18 Zusammenfassung

Objektorientierung bedeutet für uns, dass wir gewaltig umdenken müssen. Gleichzeitig ist sie aber auch ein Garant dafür, dass wir Code schreiben können, der wiederverwendbar und sehr gut wartbar ist. Setzen wir Klassen richtig ein, können wir mit Hilfe von Vererbung Funktionalitäten so kapseln, dass wir sie nur einmal definieren müssen.

Auch Pakete und Aufzählungen sind sinnvolle Sprach-Elemente von Java, erlauben sie es doch, Inhalte zu strukturieren und einfacher kontrollierbar zu halten. Und mit Hilfe von Interfaces dürfen wir Architekten spielen – wir geben nur vor, wie etwas auszusehen hat. Um die eigentliche Umsetzung dürfen sich dann Andere kümmern.

Aus eigener Erfahrung kann Ihnen der Autor bestätigen, dass objektorientiertes Denken nicht immer einfach ist – insbesondere der Einstieg fällt dem ein oder anderen besonders schwer. Sind Sie aber erst einmal in der Thematik drin, werden Sie schnell feststellen, dass Sie überhaupt nicht mehr anders arbeiten wollen – zu einfach ist es, objektorientiert zu arbeiten. Und zu gut sind die Ergebnisse, die mit diesem Ansatz erzielt werden können.