statischen Importierungen kann man sich zwar ein wenig Tipparbeit sparen, ein anderer Entwickler bezahlt aber den mehrfachen zeitlichen Aufwand für das spätere Verstehen des Quellcodes. Aus diesem Grund sollte man die statischen Importierungen grundsätzlich vermeiden. Bei mehrfach verwendeten Konstanten sind sie jedoch ein gutes Mittel zur Vermeidung des Constant Interface Antipattern.

4.3 Typsichere Aufzählungen

von Karsten Wolke

In Java gibt es zwei Varianten zur Definition von Typen: zum einen über Klassen, zum anderen über Schnittstellen. Das ist im Normalfall völlig ausreichend. Hin und wieder kommt es aber vor, daß man mit diesen Typdefinitionen auf Probleme stößt. Gibt es von einem Typ eine festgelegte Anzahl von Exemplaren, ist es zunächst schwierig, diesen Sachverhalt mit Klassen umzusetzen. Eine Klasse liefert die abstrakte Struktur eines Typs. Damit kann eine unbegrenzte endliche Anzahl von Exemplaren erzeugt werden.

Es gibt im wesentlichen zwei Varianten, mit denen Entwickler in Java Aufzählungen selbst konstruieren. Oft werden für die einzelnen Werte Konstanten definiert, meistens dann, wenn für die einzelnen Exemplare eines Typs keine Eigenschaften oder Methoden definiert werden müssen.

```
public class Marks {
  public static final int EXCELLENT   = 1;
  public static final int GOOD        = 2;
  public static final int SATISFACTORY= 3;
  public static final int SUFFICIENT  = 4;
  public static final int INSUFFICIENT= 5;
}
```

In diesem Beispiel wird jedes Exemplar der Klasse *Marks* durch einen primitiven *int*-Wert ausgedrückt. Alle Klassen, die mit dem Typ *Marks* arbeiten, drücken mit dem primitiven Datentyp *int* ein Exemplar des Typs *Marks* aus. Beispielsweise sieht eine Klasse *Student*, die einen Studenten repräsentiert, wie folgt aus:

```
public class Student {
  public String firstName;
  public String lastName;
  public int markInInformatik;
  ...
}
```

Der Variablen *markInInformatik* werden so die konstanten Werte der Klasse *Marks* zugewiesen. Bei einer sauberen Programmierung funktioniert dieses Konzept einwandfrei. Leider ist diese Variante aber grundsätzlich nicht typsicher, denn der Variablen *markInInformatik* könnte auch ein numerischer Wert außerhalb des Wertebereichs von 1 bis 5 zugewiesen werden.

Es gibt eine sauberere Variante für eine typsichere Aufzählung. Nachfolgend eine bessere Implementierung der Klasse *Marks*:

```
public class Marks2 {
  private String markName;
  public static final Marks2 EXCELLENT   = new Marks2("sehr gut");
  public static final Marks2 GOOD        = new Marks2("gut");;
```

```
  public static final Marks2 SATISFACTORY= new Marks2("befriedigend");
  public static final Marks2 SUFFICIENT  = new Marks2("ausreichend");
  public static final Marks2 INSUFFICIENT= new Marks2("mangelhaft");

  private Marks2(String name) {
    this.markName = name;
  }

  public String toString() {
    return this.markName;
  }
}
```

Diese Variante verhindert mit einem kleinen Trick, daß weitere Exemplare der Klasse erzeugt werden. Da der Konstruktor mit dem Zugriffsrecht *private* versehen ist, kann lediglich die Klasse Exemplare von sich selbst erzeugen. Diese Objekte werden gleich bei der statischen Initialisierung der Klasse erzeugt und in Konstanten der Klasse abgelegt. Es besteht somit keine Möglichkeit, weitere Exemplare der Klasse anzulegen. Die Exemplare können Methoden und Eigenschaften bereitstellen, da bei dieser Variante ein Objekt einer Klasse für die Repräsentation eines Exemplars herangezogen wird. Von der Klasse *Marks2* gibt es beispielsweise genau fünf Exemplare, die die Noten einer Hochschule darstellen. Jedes Exemplar besitzt die Methode *toString()*, die die Note als Zeichenkette zurückgibt. Bei dieser Variante könnte die Klasse *Student* wie folgt implementiert sein:

```
public class Student2 {
  public String firstName;
  public String lastName;
  public Marks2 markInInformatik;
  ...
}
```

Damit ist sichergestellt, daß wirklich nur eines der fünf Exemplare der Klasse *Marks2* der Variablen *markInInformatik* zugewiesen werden kann. Diese Variante ist zwar aufwendiger in der Implementierung als die vorherige Variante, dafür aber typsicher und wesentlich flexibler.

Um den Entwickler von einiger Arbeit zu befreien, wurden deswegen mit Java 2 SE V5 die typsicheren Aufzählungen (engl. type-safe enumerations) eingeführt. Dafür dient die neue Typstruktur *enum*. Der Java-Compiler generiert daraus eine eigene Klasse, die eine ähnliche Struktur wie die zweite soeben vorgestellte Variante besitzt.

4.3.1 Aufzählungen mit enum

Der Aufzählungstyp *enum* wurde in Java flexibel implementiert, trotzdem ist es Sun gelungen, die Definition einer Aufzählung einfach zu halten. Eine Aufzählungsdefinition besteht mindestens aus folgenden drei Angaben:

- Dem Schlüsselwort *enum*,
- einem Bezeichner für den Typ,
- einer Liste von Werten, die der Typ annehmen kann.

Optional können bei einer Aufzählungsdefinition folgende weitere Angaben gemacht werden:

- Schnittstellen, die von der Aufzählung implementiert werden.
- Variablendefinitionen.
- Konstruktorendefinitionen.
- Methodendefinitionen.
- Wertbezogene Klassenrümpfe.

Eine einfache Aufzählungsdefinition sieht wie folgt aus:

```
[<<Modifier>>] enum <<Bezeichner>>{<<Wert1>>, <<Wert2>>, ...};
```

Damit ist sie einer Klassendefinition sehr ähnlich. Statt des Schlüsselwortes *class* gibt es bei Aufzählungen das Schlüsselwort *enum*. Der Bezeichner gibt, wie bei Klassen auch, dem Typ einen Namen. In geschweiften Klammern werden die Werte des Typs angegeben. Eine *enum*-Aufzählungsdefinition für das obige Beispiel mit Noten, wie sie an Hochschulen vergeben werden, sieht wie folgt aus:

```
public enum Marks3 {EXCELLENT, GOOD, SATISFACTORY, SUFFICIENT, INSUFFICIENT};
```

Diese Anweisung definiert die vollständige Aufzählung der Noten. Sie kann wie eine Klasse behandelt werden. Dementsprechend kann diese Anweisung auch für sich in einer Textdatei mit dem Namen *Marks3.java* gespeichert werden. Der Übersetzer generiert aus dieser Anweisung eine Klasse, die eine ähnliche Struktur aufweist wie die oben in den Grundlagen vorgestellte zweite Variante.

Mit der Klasse *Student* wird der Einsatz einer Aufzählung kurz dargestellt. Überall dort, wo eine Klasse erlaubt ist, kann auch ein Typ *enum* verwendet werden. Die Klasse für Studenten wird beispielsweise wie folgt implementiert:

```
public class Student3 {
  public String firstName;
  public String lastName;
  private Marks3 markInInformatik;

  public Student3(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public void setMarkInInformatik(Marks3 mark) {
    this.markInInformatik = mark;
  }

  public Marks3 getMarkInInformatik() {
    return markInInformatik;
  }

  public String getStudentData() {
    StringBuffer buffer = new StringBuffer();
    buffer.append(firstName +" "+lastName);
    buffer.append(", Informatiknote =");
    buffer.append(markInInformatik);
    return buffer.toString();
  }
}
```

Mit dem folgenden kleinen Programm werden sowohl die Klasse *Student3* als auch die Aufzählung *Marks3* getestet:

```
public class Student3Demo {

  public static void main(String[] args) {
    Student3 meier = new Student3("Meier", "Karl");
    Student3 stenzel = new Student3("Stenzel", "Hans");
    Student3 schroeder = new Student3("Schroeder", "Frida");
    meier.setMarkInInformatik(Marks3.GOOD);
    stenzel.setMarkInInformatik(Marks3.EXCELLENT);
    schroeder.setMarkInInformatik(Marks3.SUFFICIENT);
    System.out.println(meier.getStudentData());
    System.out.println(stenzel.getStudentData());
    System.out.println(schroeder.getStudentData());
  }
}
```

Das Ergebnis auf der Konsole:

```
Meier Karl, Informatiknote =GOOD
Stenzel Hans, Informatiknote =EXCELLENT
Schroeder Frida, Informatiknote =SUFFICIENT
```

An diesem Beispiel läßt sich erkennen, wie einfach in Java seit dem JDK Version 5 Aufzählungen definiert werden. *enum*-Aufzählungen sind in Java typsicher sowie einfach und schnell umzusetzen.

4.3.2 Eigenschaften und Struktur von enum-Aufzählungen

Der Java-Compiler generiert aus einer *enum*-Aufzählungsdefinition eine Klasse. Die wichtigsten Merkmale beziehungsweise Eigenschaften von solchen Aufzählungen sind.

- *enum*-Aufzählungen sind Klassen.
  Dadurch wird die Typsicherheit gewährleistet, denn der Übersetzer kann prüfen, ob die Werte im Aufzählungstyp definiert sind. Da *enum*-Aufzählungen Klassen sind, wird auf die Elemente einer Aufzählung genau wie bei einer Klasse zugegriffen. Die Syntax ist identisch. Ein Aufzählungstyp kann überall dort verwendet werden, wo ein Klassentyp erlaubt ist.
- *enum*-Aufzählungen erben von der Klasse *java.lang.Enum*.
  Die Klasse *java.lang.Enum* wurde mit dem JDK 5 eingeführt und ist die Oberklasse aller *enum*-Aufzählungen. Sie ist selbst keine Aufzählung. Das Verhalten jeder Aufzählung ist in dieser Klasse definiert.
- *enum*-Aufzählungen haben keinen öffentlichen Konstruktor.
  Damit ist es unmöglich, Exemplare des Aufzählungstyps zu erzeugen. Es können nur die Exemplare verwendet werden, die von der Klasse selbst zur Verfügung gestellt werden.
- *enum*-Aufzählungswerte besitzen die Modifier *public*, *static* und *final*.
  Damit können die definierten Werte weder überschrieben, noch ausgewechselt oder auf eine andere Weise manipuliert werden.
- *enum* Aufzählungen sind standardmäßig *final*.
  Die Spezifikation sagt aus, daß der Entwickler bei einer *enum*-Aufzählungsdefinition die Modifier *abstract* und *final* nicht angeben darf. Dies wird vom Com-

piler automatisch übernommen. Standardmäßig sind die generierten Aufzählungsklassen mit dem Modifier *final* versehen. Das Vererben von Aufzählungen ist also nicht möglich, und somit kann man Aufzählungen nicht erweitern.

- *enum*-Aufzählungswerte können verglichen werden.
  Aufzählungswerte können mit dem Gleichheitsoperator oder mit der Methode *equals()* verglichen werden. Beide Varianten liefern exakt das gleiche Ergebnis, da die Werte einer Aufzählung statisch und konstant sind. Damit ist das Objekt, das sich hinter einem Aufzählungswert befindet, während der gesamten Programmlaufzeit gleich.
- *enum*-Aufzählungen implementieren die Schnittstelle *java.lang.Comparable*.
  Dementsprechend besitzen alle Aufzählungswerte die Methode *compareTo()*, um sie mit einem anderen Aufzählungswert zu vergleichen. Das Sortieren geschieht in der Reihenfolge, wie sie definiert wurden.
- *enum*-Aufzählungen implementieren die Schnittstelle *java.io.Serializable*.
  Damit können Aufzählungen grundsätzlich zusammen mit dem Java-Serialisations-Mechanismus verwendet werden.
- *enum*-Aufzählungen überschreiben die Methode *toString()*.
  Die Methode *toString()* gibt den Namen eines Auszählungstypen zurück. Beispielsweise liefert der Aufruf *Marks.GOOD.toString()* die Zeichenkette *GOOD*.
- *enum*-Aufzählungen besitzen eine statische Methode *valueOf()*.
  Diese Methode repräsentiert die Umkehrfunktion zur Methode *toString()*. Es wird eine Zeichenkette übergeben und der Aufzählungswert mit dem übergebenen Namen zurückgegeben. Gibt es keinen Aufzählungswert mit dem angegebenen Namen, wird eine *IllegalArgumentException* ausgelöst. Der Aufruf *Marks.valueOf("GOOD")* liefert beispielsweise den Auszählungswert *Marks.GOOD* zurück.
- *enum*-Aufzählungen besitzen eine Methode *ordinal()*.
  Die Methode *ordinal()* gibt die Position des Aufzählungswertes in der Aufzählung zurück. Es wird die Reihenfolge der Werte bei der Aufzählungsdefinition verwendet.
- *enum*-Aufzählungen besitzen eine statische Methode *values()*.
  values() liefert die Aufzählungswerte einer Aufzählung als Array zurück. Damit läßt sich eine Iteration über die Aufzählungswerte implementieren.

4.3.3 Aufzählungen innerhalb anderer Strukturen erzeugen

Wie bei Klassen auch, macht es sowohl Sinn, Aufzählungen in einer separaten Textdatei als auch innerhalb von anderen Strukturen zu erzeugen. Wird eine Aufzählung beispielsweise nur von einer Klasse verwendet, sollte die Aufzählung innerhalb der Klasse definiert werden. Dies wird auch als *Inline*-Definition bezeichnet. Der Übersetzer generiert aus einer Aufzählung innerhalb einer Struktur eine innere Klasse. Der folgende Programmcode definiert beispielsweise eine Aufzählung innerhalb einer Klasse.

```
public class Machine {
  public enum State {ONLINE, OFFLINE};
}
```

Aufzählungen, die innerhalb von anderen Strukturen definiert sind, erhalten automatisch den Modifier *static*. Das ist auch sinnvoll, da eine Aufzählung sich nicht ändert und somit für alle Exemplare der Klasse gleich aussieht. Man kann als Entwickler auch explizit den Modifier *static* angeben, dies hat jedoch keine zusätzlichen Auswirkungen.

4.3.4 Iteration über Aufzählungen

Oft kommt es vor, daß man dem Anwender eine Auswahl von *n* Werten zur Verfügung stellt. Wenn die Werte der Auswahl von Anfang an feststehen, macht es Sinn, diese Werte in einer Aufzählung zu definieren. Mit Hilfe der Methode *values()* erhält man die Werte der Aufzählung als Array. Dieses Array kann man elementweise durchlaufen und die Werte darstellen. Das folgende Programm gibt die Werte einer Aufzählung mit der *foreach*-Schleife auf der Konsole aus:

```
public class School {

  public enum Marks{EXCELLENT, GOOD, SATISFACTORY, SUFFICIENT,
                    INSUFFICIENT};

  public static void main(String[]args) {
    for (Marks mark : Marks.values()) {
      System.out.println("Note =" + mark);
    }
  }
}
```

Das Ergebnis auf der Konsole:

```
Note =EXCELLENT
Note =GOOD
Note =SATISFACTORY
Note =SUFFICIENT
Note =INSUFFICIENT
```

4.3.5 enum-Aufzählungen in switch-Anweisungen

Aufzählungen stellen eine endliche Anzahl von Werten eines Typs dar. Werden Aufzählungen in Programmen eingesetzt, kommt es oft vor, daß bestimmte Programmteile nur dann ausgeführt werden dürfen, wenn ein bestimmter Wert der Aufzählung gewählt wurde. Üblicherweise implementiert man Mehrfachverzweigungen mit der *switch*-Anweisung. Sie unterstützt für die verschiedenen Fälle die primitiven Datentypen *char*, *byte*, *short* und *int* und seit dem JDK 5 auch die Werte einer *enum*-Aufzählung. Dieser Anweisungstyp wird im Abschnitt 2.1 detailliert erläutert.

Das folgende Programm zeigt *enum*-Aufzählungen im Einsatz in *switch*-Anweisungen mit den bereits bekannten Klassen *Marks3* und *Student3*.

```
public class EnumSwitchDemo {

  public static void main(String[] args) {

    Student3 meier = new Student3("Meier", "Karl");
    Student3 stenzel = new Student3("Stenzel", "Hans");
    Student3 schroeder = new Student3("Schroeder", "Frida");
    meier.setMarkInInformatik(Marks3.GOOD);
    stenzel.setMarkInInformatik(Marks3.EXCELLENT);
    schroeder.setMarkInInformatik(Marks3.SUFFICIENT);

    printStudentData(meier);
    printStudentData(stenzel);
```

```
    printStudentData(schroeder);
  }

  private static void printStudentData(Student3 s) {
    StringBuffer buffer = new StringBuffer();
    buffer.append(s.firstName +" ");
    buffer.append(s.lastName);
    buffer.append("| Kommentar zur Veranstaltung Informatik: ");
    switch(s.getMarkInInformatik()) {
      case EXCELLENT:
        buffer.append("Besser geht es nicht !!!");
        break;
      case GOOD:
      case SATISFACTORY:
      case SUFFICIENT:
        buffer.append("Bestanden mit " + s.getMarkInInformatik());
        break;
      case INSUFFICIENT:
        buffer.append("Durchgefallen!!!");
    }
    System.out.println(buffer.toString());
  }
}
```

Das Ergebnis auf der Konsole:

```
Meier Karl| Kommentar zur Veranstaltung Informatik: Bestanden mit GOOD
Stenzel Hans| Kommentar zur Veranstaltung Informatik: Besser geht es nicht!!!
Schroeder Frida| Kommentar zur Veranstaltung Informatik: Bestanden mit SUFFICIENT
```

In der *switch*-Anweisung muß ein Aufzählungswert angegeben werden. In diesem Fall ist dies ein Aufzählungswert vom Typ *Marks3*. Besonders interessant ist die Definition der Konstanten in den *case*-Anweisungen:

```
case EXCELLENT:
case GOOD:
case SATISFACTORY:
case SUFFICIENT:
case INSUFFICIENT:
```

Man erwartet eigentlich die folgende Definition der Konstanten:

```
case Marks3.EXCELLENT:
case Marks3.GOOD:
case Marks3.SATISFACTORY:
case Marks3.SUFFICIENT:
case Marks3.INSUFFICIENT:
```

Warum auch immer, die zweite Variante der Konstantenangabe führt zu einem Übersetzungsfehler und ist damit nicht erlaubt. Die Konstanten müssen zwingend in der ersten Form angegeben werden, also ohne explizite Angabe des Typs. Dies sorgt zwar für ein schnelles Programmieren, aber der Quelltext ist schwerer zu lesen. Es bleibt abzuwarten, ob diese Schreibweise in weiteren Versionen des JDKs doch noch verbessert wird.

Wie auch bei anderen *switch*-Anweisungen müssen nicht zwingend alle Werte als Konstanten angegeben werden. Wird in ihr ein Wert entdeckt, zu dem es keine geeignete *case*-Konstante gibt, wird zu der Anweisung hinter der *switch*-Anweisung gesprungen. Eine *default*-Konstante ist erlaubt. So wird grundsätzlich immer mindestens ein Programmzweig ausgeführt. Der folgende Programmcode zeigt eine etwas andere Implementierung der Methode *printStudentData()* des vorherigen Beispiels. Bei dieser Implementierung werden nur die beste und schlechteste Note explizit über eine *case*-Konstante abgefangen. Die anderen Noten werden über *default* behandelt:

```
private static void printStudentData(Student3 s) {
  StringBuffer buffer = new StringBuffer();
  buffer.append(s.firstName +" ");
  buffer.append(s.lastName);
  buffer.append("| Kommentar zur Veranstaltung Informatik: ");
  switch(s.getMarkInInformatik()) {
    case EXCELLENT:
      buffer.append("Besser geht es nicht !!!");
      break;
    case INSUFFICIENT:
      buffer.append("Durchgefallen !!!");
      break;
    default:
      buffer.append("Bestanden mit " + s.getMarkInInformatik());
  }
  System.out.println(buffer.toString());
}
```

4.3.6 Mengen von Aufzählungen

Oft kommt es vor, daß ein Objekt mehrere Eigenschaften besitzt. Die Eigenschaften sind endlich und können deswegen in einer Aufzählung repräsentiert werden. Leider kann nicht spezifiziert werden, wie viele Eigenschaften ein Objekt besitzen kann. Viele Entwickler setzen für solche Probleme die Bitoperatoren ein. Jede Eigenschaft wird dann durch ein Bit eines ganzzahligen Datentyps repräsentiert. Der folgende Quellcode definiert einige Eigenschaften eines Autos:

```
public class CarFeaturesOld {
  public static final int DIESEL            = 1;
  public static final int BENZIN            = 2;
  public static final int AIR_CONDITIONER   = 4;
  public static final int BACK_SPOILER      = 8;
  public static final int AUTOMATIC_COUPLER =16;
}
```

Alle Eigenschaften werden durch eine Potenz von Zwei ausgedrückt und repräsentieren somit ein Bit in einem *int*-Wert. Ein Auto mit mehreren Eigenschaften wird dann wie folgt definiert:

```
int MyCarFeatures = 21
```

oder

```
int MyCarFeatures = CarFeaturesOld.DIESEL |
                    CarFeaturesOld.AIR_CONDITIONER |
                    CarFeaturesOld.AUTOMATIC_COUPLER
```

Diese Art der Programmierung war lange Zeit der einzige Weg, solche Strukturen auszudrücken. In Java gibt es seit dem JDK 5 die Klasse *java.util.EnumSet*, die für solche Strukturen herangezogen werden kann. Die Basis bildet eine *enum*-Aufzählung, die die Eigenschaften, die zur Verfügung stehen, darstellt. Für die Eigenschaften eines Autos sieht sie wie folgt aus:

```
public enum CarFeatures {
  DIESEL,
  BENZIN,
  AIR_CONDITIONER,
  BACK_SPOILER,
  AUTOMATIC_COUPLER
}
```

Ein Exemplar der Klasse *EnumSet* kann mehrere Werte einer *enum*-Aufzählung repräsentieren. Somit können mehrere Eigenschaften durch ein Objekt der Klasse *EnumSet* ausgedrückt werden. Mit deren statischen Methoden kann man ein Exemplar dieser Klasse einsetzen, Konstruktoren gibt es keine. Die wichtigsten Methoden von *EnumSet*:

```
public static EnumSet allOf(Class elementType)
```

Erzeugt ein Objekt der Klasse *EnumSet*, das alle Werte der Klasse *elementType* verwendet.

```
public static EnumSet complementOf(EnumSet s)
```

Erzeugt ein Objekt der Klasse *EnumSet*, das alle Elemente repräsentiert, die sich nicht in dem *EnumSet*-Objekt *s* befinden.

```
public static EnumSet of(E first, E... rest)
```

Erzeugt ein Objekt der Klasse *EnumSet* mit allen angegebenen Werten. Dieser Methode können beliebig viele Parameter übergeben werden. Es nutzt die Eigenschaft der variablen Parameterlisten, die mit dem JDK 5 eingeführt wurden.

```
public EnumSet clone()
```

Erzeugt eine Kopie des *EnumSet*-Objekts.

Es gibt noch eine ganze Reihe weiterer Methoden, die von *EnumSet* bereitgestellt werden. Da sie die Schnittstelle *java.util.Set* implementiert, stehen auch alle typischen Methoden der *Collection*-Klassen zur Verfügung. Mit der folgenden Anweisung wird beispielsweise ein Auto mit allen Eigenschaften definiert:

```
EnumSet myCar = EnumSet.allOf(CarFeatures);
```

Da ein Auto nicht gleichzeitig ein Diesel und ein Benziner sein kann, sollte besser die folgende Anweisung gelten, die nur einige Eigenschaften definiert:

```
EnumSet myCar = EnumSet.Of(CarFeatures.DIESEL,
                           CarFeatures.AIR_CONDITIONER,
                           CarFeatures.AUTOMATIC_COUPLER);
```

Mit Hilfe der Methode *contains()* kann man nachträglich feststellen, ob eine Eigenschaft beziehungsweise ein Wert sich in einem *EnumSet*-Objekt befindet. Die folgende Anweisung testet beispielsweise, ob das Auto ein Dieselfahrzeug ist:

```
boolean isDiesel = myCar.contains(CarFeatures.DIESEL);
```

4.3.7 Benutzerdefinierte Elemente in enums

Es wurde schon mehrfach erwähnt, daß eine *enum*-Aufzählungsdefinition zu einer Klasse generiert wird. Deswegen ist auch überall, wo eine Klasse erlaubt ist, eine Aufzählung erlaubt. Man kann sich daher eine Aufzählung als eine Klasse vorstellen, die eine bestimmte vordefinierte Struktur hat. Die Struktur einer Aufzählung kann durchaus erweitert werden. In Aufzählungen kann man (wie in Klassen) Konstruktoren, Methoden und Eigenschaften definieren. Damit werden Funktionsumfang und Einsatzgebiet von *enum*-Aufzählungen wesentlich vergrößert. Es sollten daher immer *enum*-Aufzählungen zum Einsatz kommen, wenn nur eine endliche Anzahl von Objekten eines Typs existieren dürfen.

Die Bedeutung und die Syntax von Konstruktoren, Methoden und Eigenschaften werden im Kapitel 2.2 detailliert erläutert. Sie haben bei Aufzählungen dieselbe Bedeutung und die gleiche Syntax wie für Klassen und werden deswegen hier nicht näher beschrieben. An dieser Stelle wird lediglich die Definition von Konstruktoren, Methoden und Eigenschaften innerhalb von Aufzählungen gezeigt. Zur Demonstration dient eine Aufzählung, die einige chemische Elemente darstellt:

```
public enum ChemicalElements {
  H("Wasserstoff"),
  He("Helium"),
  Li("Lithium"),
  Be("Beryllium"),
  B("Bor"),
  C("Kohlenstoff");

  private String name;

  ChemicalElements(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public double getDensity() {
    switch(this) {
      case H:  return 0.08;
      case He: return 0.17;
```

```
      case Li: return 0.53;
      case Be: return 1.85;
      case B:  return 2.46;
      case C:  return 3.51;
      default: return Double.NaN;
    }
  }
}
```

An diesem Beispiel läßt sich gut erkennen, wie Konstruktoren, Methoden und Eigenschaften in einer Aufzählung definiert werden. Nach Angabe der Werte folgt ein Semikolon und anschließend die Definition der objektorientierten Strukturen. Die Syntax von Variablendefinitionen, Konstruktoren und Methoden ist identisch zu denen einer Klasse. Nur wird ein Konstruktor bei einer Aufzählung automatisch mit dem Modifier *private* versehen, damit nicht von außen Objekte der Aufzählungsklasse erzeugt werden können. Es ist daher nicht erlaubt, explizit einen Zugriffsmodifier verschieden von *private* anzugeben.

In dem Beispiel gibt es einen Konstruktor, der als Parameter eine Zeichenkette verlangt. Alle Werte müssen deswegen mit einer Zeichenkette als Parameter angegeben werden. Die Werte der Aufzählung besitzen neben denen jeder Aufzählung die Methoden *getName()* und *getDensity()*. *getName()* gibt den Namen des chemischen Elements zurück, die Zeichenkette, die beim Konstruktor übergeben wurde. *getDensity()* liefert dagegen die Dichte des chemischen Elements in Gramm pro cm³.

Das folgende Programm gibt mit der Aufzählung *ChemicalElements* einige Daten auf der Konsole aus:

```
public class TestChemicalElements {

  public static void main(String[] args) {
    for (ChemicalElements elem : ChemicalElements.values()) {
      System.out.print("Name: " + elem.getName());
      System.out.print(" | Dichte: " + elem.getDensity());
      System.out.println(" g/cm³");
    }
  }
}
```

Das Ergebnis auf der Konsole:

```
Name: Wasserstoff | Dichte: 0.08 g/cm³
Name: Helium | Dichte: 0.17 g/cm³
Name: Lithium | Dichte: 0.53 g/cm³
Name: Beryllium | Dichte: 1.85 g/cm³
Name: Bor | Dichte: 2.46 g/cm³
Name: Kohlenstoff | Dichte: 3.51 g/cm³
```

4.3.8 Schnittstellen implementieren in Aufzählungen

Schnittstellen definieren abstrakte Methoden und können eine Art Mehrfachvererbung abbilden. Alle Klassen, die eine Schnittstelle implementieren, können als Exemplare der Schnittstelle betrachtet werden. Da Aufzählungen ebenfalls Klassen sind, können auch sie Schnittstellen implementieren. Im Vergleich zu Klassen müssen Aufzählungen jedoch direkt die Methoden der Schnittstellen implementieren. Es ist

nicht erlaubt, eine Aufzählung mit dem Modifier *abstract* zu versehen. Dies ist sinnlos, da es nicht möglich ist, daß eine Aufzählung von einer anderen Aufzählung erbt.
Es macht beispielsweise Sinn, für die physikalische Dichte eine Schnittstelle zu definieren:

```
public interface Density {
  public double getDensity();
}
```

Alle Klassen und Aufzählungen, die Objekte mit einer Dichte repräsentieren, sollten diese Schnittstelle implementieren. Dementsprechend sieht die Aufzählungsdefinition von *ChemicalElements* wie folgt aus:

```
public enum ChemicalElements implements Density {
  H("Wasserstoff"),
  He("Helium"),
  ...
}
```

Direkt nach dem Bezeichner der Aufzählung werden mit der bekannten Klausel *implements* die Schnittstellen angegeben, die die Aufzählung implementiert.

4.3.9 Aufzählungswertbezogene Klassenrümpfe

Aufzählungen in Java besitzen die Eigenschaft, daß für die einzelnen Werte spezielle Methoden definiert werden können. Das Einsatzgebiet dieser Eigenschaft ist jedoch fraglich und die Verwendung in der Java-Community umstritten, denn eine Klasse beziehungsweise eine Aufzählung definiert einen Typ. Exemplare eines Typs sollten deswegen dieselben Eigenschaften und dasselbe Verhalten aufweisen. Besitzt ein Exemplar eines Typs ein anderes Verhalten als ein anderes Exemplar, stellt sich die Frage, ob der Typ richtig gewählt wurde. Ist es nicht vielleicht geschickter, zwei Typen zu definieren?
Es scheint so, als hätte Sun diese Eigenschaft für Aufzählungen bereitgestellt, da es nicht möglich ist, daß Aufzählungen andere Aufzählungen erweitern. Man kann deswegen eine Aufzählung wie in den vorherigen Abschnitten definieren, die eine Reihe von gemeinsamen Methoden, Eigenschaften und Konstruktoren für alle Werte festlegt. Zusätzlich können abstrakte Methoden definiert werden. Diese abstrakten Methoden kann jeder Wert unterschiedlich implementieren. Ebenso kann ein Wert auch eine gemeinsame Methode überschreiben. Mit diesen Eigenschaften erweitert man sozusagen genau ein Exemplar der Aufzählung in der Aufzählungsdefinition selbst. Dies kann ausnahmsweise hilfreich sein, fördert aber die Lesbarkeit des Quelltextes nicht.
Der folgende Quelltext zeigt eine Aufzählung mit wertbezogenen Klassenrümpfen:

```
public enum ChemicalElements2 {
  H("Wasserstoff") {
    public double getDensity() {
      return 0.08;
    }

    public String getStateAtRoomTemperature() {
      return "gasfoermig";
    }
```