Leseprobe

In dieser Leseprobe gibt Christian Ullenboom einen Überblick über die Neuerungen von Java 8. Darüber hinaus erklärt er Ihnen alles über die Auszeichnungssprache XML und wie Sie diese optimal einsetzen. Außerdem können Sie einen Blick in das vollständige Inhalts- und Stichwortverzeichnis des Buches werfen.

Christian Ullenboom

Java SE 8 Standard-Bibliothek – Das Handbuch für Java-Entwickler

1.448 Seiten, gebunden, 2. Auflage 2014
49,90 Euro, ISBN 978-3-8362-2874-9

www.galileo-press.de/3607

Kapitel 1

Neues in Java 8 und Java 7

»Jede Lösung eines Problems ist ein neues Problem.«
– Johann Wolfgang von Goethe (1749–1832)

Dieses Kapitel fasst die wesentlichen Änderungen von Java 8 kompakt zusammen, sodass sich langjährige Entwickler schnell einen Überblick verschaffen können.

1.1 Sprachänderungen in Java 8

1.1.1 Statische ausprogrammierte Methoden in Schnittstellen

In der Regel deklariert eine Schnittstelle Operationen, also abstrakte Objektmethoden, die eine Klasse später implementieren muss. Die in Klassen implementierte Schnittstellenmethode kann später wieder überschrieben werden, nimmt also ganz normal an der dynamischen Bindung teil. Einen Objektzustand kann die Schnittstelle nicht deklarieren, denn Objektvariablen sind in Schnittstellen tabu – jede deklarierte Variable ist automatisch statisch, also eine Klassenvariable.

Ab Java 8 lassen sich in Schnittstellen statische Methoden unterbringen und als Utility-Methoden neben Konstanten stellen. Als statische Methoden werden sie nicht dynamisch gebunden.

Beispiel

Im vorangehenden Kapitel hatten wir schone eine Schnittstelle `Buyable` deklariert. Die Idee ist, dass alles, was käuflich ist, diese Schnittstelle implementiert und einen Preis hat. Zusätzlich gibt es eine Konstante für einen Maximalpreis:

```
interface Buyable {
  int MAX_PRICE = 10_000_000;
  double price();
}
```

Hinzufügen lässt sich nun eine statische Methode `isValidPrice(double)`, die prüft, ob sich ein Kaufpreis im gültigen Rahmen bewegt:

```
interface Buyable {
  int MAX_PRICE = 10_000_000;
  static boolean isValidPrice( double price ) {
```

```
    return price >= 0 && price < MAX_PRICE;
  }
  double price();
}
```

Von außen ist dann der Aufruf `Buyable.isValidPrice(123)` möglich.

Alle deklarierten Eigenschaften sind implizit immer `public`, sodass dieser Sichtbarkeitsmodifizierer redundant ist. Konstanten sind implizit immer statisch. Statische Methoden müssen einen Modifizierer tragen, wenn nicht, gelten sie als abstrakte Operationen.

Der Zugriff auf eine statische Schnittstellenmethode ist ausschließlich über den Namen der Schnittstelle möglich, bzw. die Eigenschaften können statisch importiert werden. Bei statischen Methoden von Klassen ist im Prinzip auch der Zugriff über eine Referenz erlaubt (wenn auch unerwünscht), etwa wie bei `new Integer(12).MAX_VALUE`. Allerdings ist das bei statischen Methoden von Schnittstellen nicht zulässig. Implementiert etwa `Car` die Schnittstelle `Buyable`, würde `new Car().isValidPrice(123)` zu einem Compilerfehler führen. Selbst `Car.isValidPrice(123)` ist falsch, was doch ein wenig verwundert, da statische Methoden normalerweise vererbt werden.

Fassen wir die erlaubten Eigenschaften einer Schnittstelle zusammen:

	Attribut	Methode
Objekt-	nein, nicht erlaubt	ja, üblicherweise abstrakt
Statische(s)	ja, als Konstante	ja, immer mit Implementierung

Tabelle 1.1 Erlaubte Eigenschaften einer Schnittstelle

Gleich werden wir sehen, dass Schnittstellenmethoden durchaus eine Implementierung besitzen können, also nicht zwingend abstrakt sein müssen.

Design

Eine Schnittstelle mit nur statischen Methoden ist ein Zeichen für ein Designproblem und sollte durch eine finale Klasse mit privatem Konstruktor ersetzt werden. Schnittstellen sind immer als Vorgaben zum Implementieren gedacht. Wenn nur statische Methoden in einer Schnittstelle vorkommen, erfüllt die Schnittstelle nicht ihren Zweck, Vorgaben zu machen, die unterschiedlich umgesetzt werden können.

1.1.2 Default-Methoden

Ist eine Schnittstelle einmal verbreitet, so sollte es dennoch möglich sein, Operationen hinzuzufügen. Java 8 bringt dafür eine Sprachänderung mit, die es Entwicklern erlaubt, neue Operationen einzuführen, ohne dass Unterklassen verpflichtet werden, diese Methoden zu implementieren. Damit das möglich ist, muss die Schnittstelle eine Standardimplementierung mitbringen. Auf diese Weise ist das Problem der »Pflicht-Implementierung« gelöst, denn wenn eine Implementierung vorhanden ist, haben die implementierenden Klassen nichts zu meckern und können bei Bedarf das Standardverhalten überschreiben. Oracle nennt diese Methoden in Schnittstellen mit vordefinierter Implementierung *Default-Methoden*.[1] Schnittstellen mit Default-Methoden heißen *erweiterte Schnittstellen*.

Eine Default-Methode unterscheidet sich syntaktisch in zwei Aspekten von herkömmlichen implizit abstrakten Methodendeklarationen:

- Die Deklaration einer Default-Methode beginnt mit dem Schlüsselwort `default`.[2]
- Statt eines Semikolons markiert bei einer Default-Methode ein Block mit der Implementierung in geschweiften Klammen das Ende der Deklaration. Die Implementierung wollen wir Default-Code nennen.

Sonst verhalten sich erweiterte Schnittstellen wie normale Schnittstellen. Eine Klasse, die eine Schnittstelle implementiert, erbt alle Operationen, sei es die abstrakten Methoden oder die Default-Methoden. Falls die Klasse nicht abstrakt sein soll, muss sie alle von der Schnittstelle geerbten abstrakten Methoden realisieren; sie kann die Default-Methoden überschreiben, muss das aber nicht, denn eine Vorimplementierung ist ja schon in der Default-Methode der Schnittstelle gegeben.

Hinweis

Erweiterte Schnittstellen bringen »Code« in eine Schnittstelle, doch das ging vorher auch schon, indem zum Beispiel eine implizite öffentliche und statische Variable auf eine Realisierung verweist:

```
interface Comparators {
  Comparator<String> TRIM_COMPARATOR = new Comparator<String>() {
    @Override public int compare( String s1, String s2 ) {
      return s1.trim().compareTo( s2.trim() );
    } };
}
```

Die Realisierung nutzt hier eine innere anonyme Klasse, ein Konzept, das genauer in Kapitel 8, »Die eXtensible Markup Language (XML)«, beleuchtet wird.

1 Der Name hat sich während der Planung für dieses Feature mehrfach gewandelt. Ganz am Anfang war der Name »defender methods« im Umlauf, dann lange Zeit »virtuelle Erweiterungsmethoden« (engl. *virtual extension methods*).

2 Am Anfang sollte `default` hinter dem Methodenkopf stehen, doch die Entwickler wollten `default` so wie einen Modifizierer wirken lassen; da Modifizierer aber am Anfang stehen, rutschte auch `default` nach vorne. Eigentlich ist ein Modifizierer auch gar nicht nötig, denn wenn es eine Implementierung, also einen Codeblock, in `{}` gibt, ist klar, dass es eine Default-Methode wird. Doch die Entwickler wollten eine explizite Dokumentation, so wie auch `abstract` eingesetzt wird – auch dieser Modifizierer bei Methoden wäre eigentlich gar nicht nötig, denn gibt es keinen Codeblock, wenn eine Methode immer abstrakt ist.

1.1.3 Erweiterte Schnittstellen deklarieren und nutzen

Realisieren wir dies in einem Beispiel. Für Spielobjekte soll ein Lebenszyklus möglich sein; der besteht aus `start()` und `finish()`. Der Lebenszyklus ist als Schnittstelle vorgegeben, die Spielobjektklassen implementieren können. Version 1 der Schnittstelle sieht also so aus:

```
interface GameLifecycle {
  void start();
  void finish();
}
```

Klassen wie `Player`, `Room`, `Door` können die Schnittstelle erweitern, und wenn sie dies tun, müssen sie die beiden Methoden implementieren. Bei Spielobjekten, die diese Schnittstelle implementieren, kann unser Hauptprogramm, das Spiel, diese Methoden aufrufen und den Spielobjekten Rückmeldung geben, ob sie gerade in das Spiel gebracht wurden oder ob sie aus dem Spiel entfernt wurden.

Je länger Software lebt, desto mehr bedauern Entwickler Designentscheidungen. Die Umstellung einer ganzen Architektur ist eine Mammutaufgabe, einfache Änderungen wie das Umbenennen sind über ein Refactoring schnell erledigt. Nehmen wir an, dass es auch bei unserer Schnittstelle einen Änderungswunsch gibt – nur die Initialisierung und das Ende zu melden, reicht nicht. Geht das Spiel in einen Pausemodus, soll ein Spielobjekt die Möglichkeit bekommen, im Hintergrund laufende Programme anzuhalten. Das soll durch eine zusätzliche `pause()`-Methode in der Schnittstelle realisiert werden. Hier spielen uns Default-Methoden perfekt in die Hände, denn wir können die Schnittstelle erweitern, aber eine leere Standardimplementierung mitgeben. So müssen Unterklassen die `pause()`-Methode nicht implementieren, können dies aber; Version 2 der nun erweiterten Schnittstelle `GameLifecycle`:

```
interface GameLifecycle {
  void start();
  void finish();
  default void pause() {}
}
```

Klassen, die `GameLifecycle` schon genutzt haben, bekommen von der Änderung nichts mit. Der Vorteil: Die Schnittstelle kann sich weiterentwickeln, aber alles bleibt binärkompatibel, und nichts muss neu compiliert werden. Vorhandener Code kann auf die neue Methode zurückgreifen, die automatisch mit der »leeren« Implementierung vorhanden ist. Weiterhin verhalten sich Default-Methoden wie andere Methoden von Schnittstellen auch: Es bleibt bei der dynamischen Bindung, wenn implementierende Klassen die Methoden überschreiben. Wenn eine Unterklasse wie `Flower` zum Beispiel bei der Spielpause nicht mehr blühen möchte, so überschreibt sie die Methode und lässt etwa den Timer pausieren. Eine Tür dagegen hat nichts zu stoppen und kann mit dem Default-Code in `pause()` gut leben. Das Vorgehen ist ein wenig vergleichbar mit normalen nichtfinalen Methoden: Sie können, müssen aber nicht überschrieben werden.

Hinweis

Statt des leeren Blocks könnte der Rumpf auch `throw new UnsupportedOperationException("Not yet implemented");` beinhalten, um anzukündigen, dass es keine Implementierung gibt. So führt eine hinzugenommene Default-Methode zwar zu keinem Compilerfehler, aber zur Laufzeit führen nicht überschriebene Methoden zu einer Ausnahme. Erreicht ist das Gegenteil vom Default-Code, weil eben keine Logik standardmäßig ausgeführt wird; das Auslösen einer Ausnahme zum Melden eines Fehlers wollen wir nicht als Logik ansehen.

Kontext der Default-Methoden

Default-Methoden verhalten sich wie Methoden in abstrakten Klassen und können alle Methoden der Schnittstelle (inklusive der geerbten Methoden) aufrufen.[3] Die Methoden werden später dynamisch zur Laufzeit gebunden.

Nehmen wir eine Schnittstelle `Buyable` für käufliche Objekte:

```
interface Buyable {
  double price();
}
```

Leider schreibt die Schnittstelle nicht vor, ob Dinge überhaupt käuflich sind. Eine Methode wie `hasPrice()` wäre in `Buyable` ganz gut aufgehoben. Was kann aber die Default-Implementierung sein? Wir können auf `price()` zurückgreifen und testen, ob die Rückgabe ein gültiger Preis ist. Das soll gegeben sein, wenn der Preis echt größer 0 ist.

```
interface Buyable {
  double price();
  default boolean hasPrice() { return price() > 0; }
}
```

Implementieren Klassen die Schnittstelle `Buyable`, müssen sie `price()` implementieren, wenn die Methode keine Default-Methode ist. Doch es ist ihnen freigestellt `hasPrice()` zu überschreiben, mit eigener zu Logik füllen und nicht die Default-Implementierung zu verwenden. Wenn implementierende Klassen keine neue Implementierung wählen, bekommen sie den Default-Code und erben eine konkrete Methode `hasPrice()`. In dem Fall geht ein Aufruf von `hasPrice()` intern weiter an `price()` und dann genau an die Klasse, die `Buyable` und die Methode `price()` implementiert. Die Aufrufe sind dynamisch gebunden und landen bei der tatsächlichen Implementierung.

3 Und damit lässt sich das bekannte Template-Design-Pattern realisieren.

Hinweis

Eine Schnittstelle kann die Methoden der absoluten Oberklasse `java.lang.Object` ebenfalls deklarieren, etwa um mit Javadoc eine Beschreibung hinzuzufügen. Allerdings ist es *nicht* möglich, mittels Default-Code Methoden wie `toString()` oder `hashCode()` vorzubelegen.

Neben der Möglichkeit, auf Methoden der eigenen Schnittstelle zurückzugreifen, steht auch die `this`-Referenz zur Verfügung. Das ist sehr wichtig, denn so kann der Default-Code an Utility-Methoden delegieren und einen Verweis auf sich selbst übergeben. Hätten wir zum Beispiel schon eine `hasPrice(Buyable)`-Methode in einer Utility-Klasse `PriceUtils` implementiert, so könnte der Default-Code aus einer einfachen Delegation bestehen:

```
class PriceUtils {
  public static boolean hasPrice( Buyable b ) { return b.price() > 0; }
}
interface Buyable {
  double price();
  default boolean hasPrice() { return PriceUtils.hasPrice( this ); }
}
```

Dass die Methode `PriceUtils.hasPrice(Buyable)` für den Parameter den Typ `Buyable` vorsieht und sich der Default-Code mit `this` auf genauso ein `Buyable`-Objekt bezieht, ist natürlich kein Zufall, sondern bewusst gewählt. Der Typ der `this`-Referenz zur Laufzeit entspricht dem der Klasse, die die Schnittstelle implementiert hat und deren Objektexemplar gebildet wurde.

Haben die Default-Methoden weitere Parameter, so lassen sich auch diese an die statische Methode weiterreichen:

```
class PriceUtils {
  public static boolean hasPrice( Buyable b ) { return b.price() > 0; }
  public static double defaultPrice( Buyable b, double defaultPrice ) {
    if ( b != null && b.price() > 0 )
      return b.price();
    return defaultPrice;
  }
}
interface Buyable {
  double price();
  default boolean hasPrice() { return PriceUtils.hasPrice( this ); }
  default double defaultPrice( double defaultPrice ) {
   return PriceUtils.defaultPrice( this, defaultPrice );
  }
}
```


Da Schnittstellen auch statische Utility-Methoden mit Implementierung enthalten können, kann der Default-Code auch hier weiterleiten. Allerdings sind die statischen Schnittstellen-Methoden immer öffentlich, und vielleicht möchte der Default-Code an eine geschützte paketsichtbare Methode weiterleiten. Außerdem ist es vorzuziehen, die Implementierung auszulagern, um die Schnittstellen nicht so codelastig werden zu lassen. Nutzt das JDK Default-Code, so gibt es in der Regel immer eine statische Methode in einer Utility-Klasse.

1.1.4 Erweiterte Schnittstellen, Mehrfachvererbung und Mehrdeutigkeiten *

Hintergrund zur Einführung von Default-Methoden war die Notwendigkeit, Schnittstellen im Nachhinein ohne nennenswerte Compilerfehler mit neuen Operationen ausstatten zu können. Ideal ist, wenn neue Default-Methoden hinzukommen und Standardverhalten definieren, und es dadurch zu keinem Compilerfehler für implementierende Klassen kommt oder zu Fehlern bei Schnittstellen, die erweiterte Schnittstellen erweitern.

Erweiterte Schnittstellen mit Default-Code nehmen ganz normal an der objektorientierten Modellierung teil, können vererbt und überschrieben werden und werden dynamisch gebunden. Nun gibt es einige Sonderfälle, die wir uns anschauen müssen. Es kann vorkommen, dass zum Beispiel

- eine Klasse von einer Oberklasse eine Methode erbt, aber gleichzeitig auch von einer Schnittstelle Default-Code für die gleiche Methode oder
- eine Klasse von zwei erweiterten Schnittstellen unterschiedliche Implementierungen angeboten bekommt.

Gehen wir verschiedene Fälle durch.

Überschreiben von Default-Code

Eine Schnittstelle kann andere Schnittstellen erweitern und neuen Default-Code bereitstellen. Mit anderen Worten: Default-Methoden können andere Default-Methoden aus Oberschnittstellen überschreiben und mit neuem Verhalten implementieren.

Führen wir eine Schnittstelle `Priced` mit einer Default-Methode ein:

```
interface Priced {
  default boolean hasPrice() { return true; }
}
```

Eine andere Schnittstelle kann die Default-Methode überschreiben:

```
interface NotPriced extends Priced {
  @Override default boolean hasPrice() { return false; }
}

public class TrueLove implements NotPriced {
  public static void main( String[] args ){
```

```
    System.out.println( new TrueLove().hasPrice() );          // false
  }
}
```

Implementiert die Klasse `TrueLove` die Schnittstelle `NotPriced`, so ist alles in Ordnung und es entsteht kein Konflikt. Die Vererbungsbeziehung ist linear `TrueLove` → `NotPriced` → `Priced`.

Klassenimplementierung geht vor Default-Methoden

Implementiert eine Klasse eine Schnittstelle und erbt außerdem von einer Oberklasse, kann Folgendes passieren: Die Schnittstelle hat Default-Code für ein Methode, und die Oberklasse vererbt ebenfalls die gleiche Methode mit Code. Dann bekommt die Unterklasse von zwei Seiten eine Implementierung. Zunächst muss der Compiler entscheiden, ob so etwas überhaupt syntaktisch korrekt ist. Ja, das ist es!

```
interface Priced {
  default boolean hasPrice() { return true; }
}

class Unsaleable {
  public boolean hasPrice() { return false; }
}

public class TrueLove extends Unsaleable implements Priced {
  public static void main( String[] args ) {
    System.out.println( new TrueLove().hasPrice() );   // false
  }
}
```

`TrueLove` erbt die Implementierung `hasPrice()` von der Oberklasse `Unsaleable` und auch von der erweiterten Schnittstelle `Buyable`. Der Code compiliert und führt zu der Ausgabe `false` – die Klasse mit dem Code »gewinnt« also gegen den Default-Code. Merken lässt sich das ganz einfach an der Reihenfolge `class` … `extends` … `implements` … – es steht `extends` am Anfang, also haben Methoden aus Implementierungen hier eine höhere Priorität als die aus erweiterten Schnittstellen.

Default-Methoden aus speziellen Oberschnittstellen ansprechen *

Eine Unterklasse kann eine konkrete Methode der Oberklasse überschreiben, aber dennoch auf die Implementierung der überschriebenen Methode zugreifen. Allerdings muss der Aufruf über `super` erfolgen, da sich sonst ein Methodenaufruf rekursiv verfängt.

Default-Methoden können andere Default-Methoden aus Oberschnittstellen ebenfalls überschreiben und mit neuem Verhalten implementieren. Doch genauso wie normale Methoden können sie mit `super` auf Default-Verhalten aus dem übergeordneten Typ zurückgreifen.

Nehmen wir für ein Beispiel unsere bekannte Schnittstelle `Buyable` und eine neue erweiterte Schnittstelle `PeanutsBuyable` an:

```
interface Buyable {
  double price();
  default boolean hasPrice() { return price() > 0; }
}

interface PeanutsBuyable extends Buyable {
  @Override default boolean hasPrice() {
    return Buyable.super.hasPrice() && price() < 50_000_000;
  }
}
```

In der Schnittstelle `Buyable` sagt der Default-Code von `hasPrice()` aus, dass alles einen Preis hat, das größer als 0 ist. `PeanutsBuyable` dagegen nutzt eine erweiterte Definition und implementiert daher das Default-Verhalten neu. Nach den berühmten kopperschen Peanuts[4] ist alles unter 50 Millionen problemlos käuflich und verursacht – zumindest für die Deutsche Bank – keine Schmerzen. In der Implementierung von `hasPrice()` greift `PeanutsBuyable` auf den Default-Code von `Buyable` zurück, um vom Obertyp eine Entscheidung über die Preiseigenschaft zu bekommen, die aber mit der Und-Verknüpfung noch spezialisiert wird.

Default-Code für eine Methode von mehreren Schnittstellen erben *

Wenn eine Klasse aus zwei erweiterten Schnittstellen den gleichen Default-Code angeboten bekommt, führt das zu einem Compilerfehler. Die Klasse `RockAndRoll` zeigt dieses Dilemma:

```
interface Sex {
  default boolean hasPrice() { return false; }
}

interface Drugs {
  default boolean hasPrice() { return true; }
}
public class RockAndRoll implements Sex, Drugs { } // Compilerfehler
```

Selbst wenn beide Implementierungen identisch wären, müsste der Compiler das ablehnen, denn der Code könnte sich ja jederzeit ändern.

Mehrfachvererbungsproblem mit super lösen

Die Klasse `RockAndRoll` lässt sich so nicht übersetzen, weil die Klasse aus zwei Quellen Code bekommt. Das Problem kann aber einfach gelöst werden, indem in `RockAndRoll` die `hasPrice()`-Me-

4 *http://de.wikipedia.org/wiki/Hilmar_Kopper#.E2.80.9EPeanuts.E2.80.9C*

thode überschrieben und dann an eine Methode delegiert wird. Um rekursive Aufrufe zu vermeiden, kommt wieder `super` mit der neuen Schreibweise ins Spiel:

```
interface Sex {
  default boolean hasPrice() { return false; }
}

interface Drugs {
  default boolean hasPrice() { return true; }
}

class RockAndRoll implements Sex, Drugs {
  @Override public boolean hasPrice() { return Sex.super.hasPrice(); }
}
```

Abstrakte überschriebene Schnittstellenoperationen nehmen Default-Methoden weg

Default-Methoden haben eine interessante Eigenschaft, dass Untertypen den Status von »haben Implementierung« auf »habe keine Implementierung« ändern können:

```
interface Priced {
  default boolean hasPrice() { return false; }
}
interface Buyable extends Priced {
  @Override boolean hasPrice();
}
```

Die Schnittstelle `Priced` bietet eine Default-Methode. `Buyable` erweitert die Schnittstelle `Priced`, aber überschreibt die Methode – jedoch nicht mit Code! Dadurch wird sie in `Buyable` abstrakt. Eine abstrakte Methode kann also durchaus eine Default-Methode überschreiben. Klassen, die `Buyable` implementieren, müssen also weiterhin eine `hasPrice()`-Methode implementieren, wenn sie nicht selbst abstrakt sein wollen. Es ist schon ein interessantes Java-Feature, dass die Implementierung einer Default-Methode in einem Untertyp wieder »weggenommen« werden kann. Bei der Sichtbarkeit ist das zum Beispiel nicht möglich: Ist eine Methode einmal öffentlich, kann eine Unterklasse die Sichtbarkeit nicht einschränken.

Das Verhalten vom Compiler hat einen großen Vorteil: Bestimmte Veränderungen der Oberschnittstelle sind erlaubt und haben keine Auswirkungen auf die Untertypen. Nehmen wir an, `hasPrice()` hätte es in `Priced` vorher nicht gegeben, sondern nur abstrakt in `Buyable`. Default-Code ist ja nur eine nette Geste, und diese sollte schmerzlos in `Priced` integriert werden können. Anders gesagt: Entwickler können in den Basistyp so eine Default-Methode ohne Probleme aufnehmen, ohne dass es in den Untertypen zu Fehlern kommt. Obertypen lassen sich also ändern, ohne die Untertypen anzufassen. Im Nachhinein kann aber zur Dokumentation die Annotation `@Override` an die Unterschnittstelle gesetzt werden.

Nicht nur eine Unterschnittstelle kann die Default-Methoden »wegnehmen«, sondern auch eine abstrakte Klasse:

```
abstract class Food implements Priced {
  @Override public abstract double price();
}
```

Die Schnittstelle `Priced` bringt eine Default-Methode mit, doch die abstrakte Klasse `Buyable` nimmt diese wieder weg, sodass erweiternde `Buyable`-Klassen auf jeden Fall `price()` implementieren müssen, wenn sie nicht selbst `abstract` sein wollen.

1.1.5 Bausteine bilden mit Default-Methoden *

Default-Methoden geben Bibliotheksdesignern ganz neue Möglichkeiten. Heute ist noch gar nicht richtig abzusehen, was Entwickler damit machen werden und welche Richtung die Java-API einschlagen wird. Auf jeden Fall wird sich die Frage stellen, ob eine Standardimplementierung als Default-Code in eine Schnittstelle wandert oder wie bisher eine Standardimplementierung als abstrakte Klasse bereitgestellt wird, von der wiederum andere Klassen ableiten. Als Beispiel sei auf die Datenstrukturen verwiesen: Eine Schnittstelle `Collection` schreibt Standardverhalten vor, `AbstractCollection` gibt eine Implementierung soweit wie möglich vor, und Unterklassen wie Listen setzen dann noch einmal auf diese Basisimplementierung auf. Erweiterte Schnittstellen können Hierarchien abbauen, denn auf eine abstrakte Basisimplementierung kann verzichtet werden. Auf der anderen Seite kann aber eine abstrakte Klasse einen Zustand über Objektvariablen einführen, was eine Schnittstelle nicht kann.

Default-Methoden können aber noch etwas ganz anderes: Sie können als Bauelemente für Klassen dienen. Eine Klasse kann mehrere Schnittstellen mit Default-Methoden implementieren und erbt im Grunde damit Basisfunktionalität von verschiedenen Stellen. In anderen Programmiersprachen ist das als Mixin bzw. Trait bekannt. Das ist ein Unterschied zur Mehrfachvererbung, die in Java nicht zulässig ist. Schauen wir uns diesen Unterschied jetzt einmal genauer an.

Default-Methoden zur Entwicklung von Traits nutzen

Was ist das Kernkonzept der objektorientierten Programmierung? Wohl ohne zu Zögern können wir Klassen, Kapselung und Abstraktion nennen. Klassen und Klassenbeziehungen sind das Gerüst eines jeden Java-Programms. Bei der Vererbung wissen wir, dass Unterklassen Spezialisierungen sind und das liskovsche Substitutionsprinzip gilt: Falls ein Typ gefordert ist, können wir auch einen Untertyp übergeben. So sollte perfekte Vererbung aussehen: Eine Unterklasse spezialisiert das Verhalten, aber erbt nicht einfach von einer Klasse, weil sie nützliche Funktionalität hat. Aber warum eigentlich nicht? Als Erstes ist zu nennen, dass das Erben aufgrund der Nützlichkeit oft gegen die Ist-eine-Art-von-Beziehung verstößt und dass uns Java zweitens nur Einfachvererbung mit nur einer einzigen Oberklasse erlaubt. Wenn eine Klasse etwas Nützliches wie Logging anbietet und unsere Klasse davon erbt, kann sie nicht gleichzeitig von einer anderen Klasse erben, um zum Beispiel Zustände in Konfigurationsdaten festzuhalten. Eine unglück-

liche Vererbung verbaut also eine spätere Erweiterung. Das Problem bei der »Funktionalitätsvererbung« ist also, dass wir uns nur einmal festlegen können.

Wenn eine Klasse eine gewisse Funktionalität einfach braucht, woher soll sie denn dann kommen, wenn nicht aus der Oberklasse? Eigentlich gibt es hier nur eine naheliegende Variante: Die Klasse greift auf andere Objekte per Delegation zurück. Wenn ein Punkt mit Farbe nicht von `java.awt.Point` erben soll, kann ein Farbpunkt einfach in einer internen Variablen einen `Point` referenzieren. Das ist eine Lösung, aber dann nicht optimal, wenn eine Ist-eine-Art-von-Beziehung besteht. Und Schnittstellen wurden ja gerade eingeführt, damit eine Klasse mehrere Typen besitzt. Abstraktionen über Schnittstellen und Oberklassen sind wichtig, und Delegation hilft hier nicht. Gewünscht ist eine Technik, die einen Programmbaustein in eine Klasse setzen kann – im Grunde so etwas wie Mehrfachvererbung, aber doch anders, weil die Bausteine nicht als komplette Typen auftreten, der Baustein selbst ist nur ein Implantat und alleine uninteressant. Auch ein Objekt kann von diesem Bausteintyp nicht erzeugt werden.

Am ehesten sind die Bausteine mit abstrakten Klassen vergleichbar, doch das wären Klassen, und Nutzer könnten nur einmal von diesem Baustein erben. Mit Java 8 gibt es aber eine ganz neue Möglichkeit, und zwar mit den erweiterten Schnittstellen: Sie bilden die Bausteine, von denen Klassen Funktionalität bekommen können. Andere Programmiersprachen bieten so etwas Ähnliches, und das Konzept wird dort Mixin oder Trait genannt.[5] Diese Bausteine sind nützlich, denn so lässt sich ein Algorithmus in eine extra Kompilationseinheit setzen und leichter wiederverwenden. Ein Beispiel: Nehmen wir zwei erweiterte Schnittstellen `PersistentPreference` und `Logged` an. Die erste erweiterte Schnittstelle soll mit `store()` Schlüssel-Wert-Paare in die zentrale Konfiguration schreiben, und `get()` soll sie auslesen:

```
import java.util.prefs.Preferences;

interface PersistentPreference {

  default void store( String key, String value ) {
    Preferences.userRoot().put( key, value );
  }

  default String get( String key ) {
    return Preferences.userRoot().get( key, "" );
  }
}
```

Die zweite erweiterte Schnittstelle ist `Logged` und bietet uns drei kompakte Logger-Methoden:

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

interface Logged {
```

5 Siehe etwa *http://scg.unibe.ch/archive/papers/Scha02aTraitsPlusGlue2002.pdf*.


```
  default void error( String message ) {
    Logger.getLogger( getClass().getName() ).log( Level.SEVERE, message );
  }

  default void warn( String message ) {
    Logger.getLogger( getClass().getName() ).log( Level.WARNING, message );
  }

  default void info( String message ) {
    Logger.getLogger( getClass().getName() ).log( Level.INFO, message );
  }
}
```

Eine Klasse kann diese Bausteine nun einbauen:

```
class Player implements PersistentPreference, Logged {
  // …
}
```

Die Methoden sind nun Teil vom `Player` und können auch von Unterklassen überschrieben werden. Als Aufgabe für den Leser bleibt, die Implementierung von `store()` im `Player` zu verändern, sodass der Schlüssel immer mit »player.« beginnt. Die Frage, die der Leser beantworten sollte, ist, ob `store()` von `Player` auf das `store()` von der erweiterten Schnittstelle zugreifen kann.

Default-Methoden weiter gedacht

Für diese Bausteine, also die erweiterten Schnittstellen, gibt es viele Anwendungsfälle. Da die Java-Bibliothek schon an die 20 Jahre alt ist, würden heute einige Typen anders aussehen. Dass sich Objekte mit `equals()` vergleichen lassen können, könnte heute zum Beispiel in einer erweiterten Schnittstelle stehen, etwa so: `interface Equals { boolean equals( Object that ) default { return this == that; } }`. So müsste `java.lang.Object` die Methode nicht für alle vorschreiben, wobei das jetzt sicherlich kein Nachteil ist. Natürlich gilt das Gleiche auch für die `hashCode()`-Methode, die heutzutage aus einer erweiterten Schnittstelle `Hashable` stammen könnte.

Und `java.lang.Number` ist ein weiteres Beispiel. Die abstrakte Basisklasse für Werte repräsentierende Objekte deklariert die abstrakten Methoden `doubleValue()`, `floatValue()`, `intValue()`, `longValue()` und die konkreten Methoden `byteValue()` und `shortValue()`. Bisher erben `AtomicInteger`, `AtomicLong`, `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long`, `Short` von dieser Oberklasse. Auch diese Funktionalität ließe sich mit einer erweiterten Schnittstelle umsetzen.

Zustand in den Bausteinen?

Nicht jeder wünschenswerte Baustein ist mit erweiterten Schnittstellen möglich. Ein Grund ist, dass die Schnittstellen keinen Zustand einbringen können. Nehmen wir zum Beispiel einen Container als Datenstruktur, der Elemente aufnimmt und verwaltet. Einen Baustein für einen

Container können wir nicht so einfach implementieren, da ein Container Kinder verwaltet, und hierfür ist eine Objektvariable für den Zustand nötig. Schnittstellen haben nur statische Variablen, und die sind für alle sichtbar; und selbst wenn die Schnittstelle eine modifizierbare Datenstruktur referenzieren würde, würde jeder Nutzer des Container-Bausteins von den Veränderungen betroffen sein. Da es keinen Zustand gibt, existieren auch für Schnittstellen keine Konstruktoren und folglich auch nicht für solche Bausteine. Denn wo es keinen Zustand gibt, gibt es auch nichts zu initialisieren. Wenn eine Default-Methode einen Zustand benötigt, muss sie selbst diesen Zustand erfragen. Hier lässt sich eine Technik einsetzen, die Oracles Java Language Architect Brian Goetz »virtual field pattern«[6] nennt. Wie das geht, zeigt das folgende Beispiel.

Referenziert ein Behälter eine Menge von Objekten, die sortierbar sind, können wir einen Baustein `Sortable` mit einer Methode `sort()` realisieren. Die Schnittstelle `Comparable` soll die Klasse nicht direkt implementieren, da ja nur die referenzierten Elemente sortierbar sind, nicht aber Objekte der Klasse selbst, und zudem soll weiterhin eine neue Methode `sort()` hinzukommen. Damit das Sortieren gelingt, muss die Implementierung irgendwie an die Daten gelangen, und hier kommt ein Trick ins Spiel: Zwar ist `sort()` eine Default-Methode, doch die erweiterte Schnittstelle besitzt Methoden, die die Klasse implementieren muss, die dem Sortierer die Daten geben. Im Quellcode sieht das so aus:

Listing 1.1 SortableDemo.java, Teil 1

```
import java.util.*;

interface Sortable<T extends Comparable<?>> {

  T[] getValues();

  void setValues( T[] values );

  default void sort() {
    T[] values = getValues();
    Arrays.sort( values );
    setValues( values );
  };
}
```

Damit `sort()` an die Daten kommt, erwartet `Sortable` von den implementieren Klassen eine Methode `getValues()`. Und damit die Daten nach dem Sortieren wieder zurückgeschrieben werden können, eine zweite Methode `setValues(…)`. Der Clou ist, dass die spätere Implementierung von `Sortable` mit den beiden Methoden dem Sortierer Zugriff auf die Daten gewährt – allerdings auch jedem anderem Stück Code, da die Methoden öffentlich sind. Da bleibt ein unschönes »Geschmäckle« zurück.

6 *http://mail.openjdk.java.net/pipermail/lambda-dev/2012-July/005171.html*

Ein Nutzer von `Sortable` soll `RandomValues` sein; die Klasse erzeugt intern Zufallszahlen.

Listing 1.2 SortableDemo.java, Teil 2

```
class RandomValues implements Sortable<Integer> {

  private List<Integer> values = new ArrayList<>();

  public RandomValues() {
    Random r = new Random();
    for ( int i = r.nextInt( 20 ) + 1; i > 0; i-- )
      values.add( r.nextInt(10000) );
  }

  @Override public Integer[] getValues() {
    return values.toArray( new Integer[values.size()] );
  }

  @Override public void setValues( Integer[] values ) {
    this.values.clear();
    Collections.addAll( this.values, values );
  }
}
```

Damit sind die Typen vorbereitet, und eine Demo schließt das Beispiel ab:

Listing 1.3 SortableDemo.java, Teil 3

```
public class SortableDemo {
  public static void main( String[] args ) {
    RandomValues r = new RandomValues();
    System.out.println( Arrays.toString( r.getValues() ) );
    r.sort();
    System.out.println( Arrays.toString( r.getValues() ) );
  }
}
```

Aufgerufen kommt auf die Konsole zum Beispiel:

```
[2732, 4568, 4708, 4302, 4315, 5946, 2004]
[2004, 2732, 4302, 4315, 4568, 4708, 5946]
```

So interessant diese Möglichkeit auch ist, ein Problem wurde schon angesprochen: Jede Methode in einer Schnittstelle ist `public`, ob sie nun eine abstrakte oder Default-Methode ist. Es wäre schön, wenn die Datenzugriffsmethoden nicht öffentlich sind, aber das geht nicht.

Wo wir gerade bei der Sichtbarkeit sind. Gibt es im Default-Code Codeduplizierung, so kann der gemeinsame Code bisher nicht in private Methoden ausgelagert werden, da es private Operationen in Schnittstellen nicht gibt. Allerdings läuft gerade ein Test, ob so etwas eingeführt werden soll.

[!]

Warnung!

Natürlich lässt sich mit Rumgetrickse ein Speicherort finden, der Exemplarzustände speichert. Es lässt sich zum Beispiel in der Schnittstelle ein Assoziativspeicher referenzieren, der eine `this`-Instanz mit einem Objekt assoziiert. Ein Container-Baustein, der mit `add()` Objekte in eine Liste setzt und sie mit `iterable()` herausgibt, könnte so aussehen:

```
interface ListContainer<T> {
  Map<Object,List<Object>> $ = new HashMap<>();
  default void add( T e ) {
    if ( ! $.containsKey( this ) )
      $.put( this, new ArrayList<Object>() );
    $.get( this ).add( e );
  }
  default public Iterable<T> iterable() {
    if ( ! $.containsKey( this ) )
      return Collections.emptyList();
    return (Iterable<T>) $.get( this );
  }
}
```

Nicht nur die öffentliche Konstante `$` ist ein Problem, sondern auch, dass es ein großartiges doppeltes Speicherloch ist. Ein Exemplar der Klasse, die diese erweiterte Schnittstelle nutzt, kann nicht so einfach entfernt werden, denn in der Sammlung ist noch eine Referenz auf das Objekt, und diese Referenz verhindert eine automatische Speicherbereinigung. Selbst wenn dieses Objekt weg wäre, hätten wir noch all die referenzierten Kinder der Sammlung in der `Map`. Das Problem ist nicht wirklich zu lösen, und hier müsste mit schwachen Referenzen tief in die Java-Voodoo-Kiste gegriffen werden. Alles in allem, keine gute Idee, und Java-Chefentwickler Brian Goetz macht auch klar:

> *»Please don't encourage techniques like this. There are a zillion ›clever‹ things you can do in Java, but shouldn't. We knew it wouldn't be long before someone suggested this, and we can't stop you. But please, use your power for good, and not for evil. Teach people to do it right, not to abuse it.«*[7]

Daher: Es ist eine schöne Spielerei, aber der Zustand sollte eine Aufgabe der abstrakten Basisklassen oder vom Delegate sein.

7 *http://mail.openjdk.java.net/pipermail/lambda-dev/2012-July/005166.html*

Zusammenfassung

Was wir in den letzten Beispielen zu den Bausteinen gemacht haben, war, ein Standardverhalten in Klassen einzubauen, ohne dass dabei der Zugriff auf die nur einmal existierende Basisklasse nötig war und ohne dass die Klasse an Hilfsklassen delegiert. In dieser Arbeitsweise können Unterklassen in jedem Fall die Methoden überschreiben und spezialisieren. Wir haben es also mit üblichen Klassen zu tun und mit erweiterten Schnittstellen, die nicht selbst eigenständige Entitäten bilden. In der Praxis wird es immer Fälle geben, in denen für eine Umsetzung eines Problems entweder eine abstrakte Klasse oder eine erweiterte Schnittstelle in Frage kommt. Wir sollten uns dann noch einmal an die Unterschiede erinnern: Eine abstrakte Klasse kann Methoden aller Sichtbarkeiten haben und sie auch final setzen, sodass sie nicht mehr überschrieben werden können. Eine Schnittstelle dagegen ist mit puren virtuellen und öffentlichen Methoden darauf ausgelegt, dass die Implementierung überschrieben werden kann.

1.2 Lambda-Ausdrücke und funktionale Programmierung

Bei der Entwicklung von Maschinensprache (bzw. Assembler) hin zur Hochsprache ist eine interessante Geschichte der Parametrisierung abzulesen. Schon die ersten Hochsprachen erlaubten eine Parametrisierung von Funktionen mit unterschiedlichen Argumenten. Die Programmiersprache Java, die im Jahr 1996 geboren wurde, bot das von Anfang an, da sie erst mehrere Jahrzehnte nach den ersten Hochsprachen entstand. Relativ spät folgten dann die Generics. Die Parametrisierung des Typs wurde erst 2004 mit der Version 5 realisiert. Bis dahin konnte eine Liste zum Beispiel Zeichenketten ebenso enthalten wie Zwerge (als Java-Objekte). Funktionale Programmierung ermöglichte nun eine Parametrisierung des Verhaltens; eine Sortiermethode arbeitet immer gleich, aber ihr Verhalten bei den Vergleichen wird angepasst. Das ist eine ganz andere Qualität, als unterschiedliche Werte zu übergeben. Das bietet nun seit 2014 die Version Java 8 elegant und einfach mit den so genannten Lambda-Ausdrücken.

1.2.1 Code = Daten

Wer den Begriff »Daten« hört, denkt zunächst einmal an Zahlen, Bytes, Zeichenketten oder auch komplexe Objekte mit ihrem Zustand. Wir wollen in diesem Kapitel diese Sicht ein wenig erweitern und auf Programmcode lenken. Java-Code, versinnbildlicht als Serie von Bytecodes, besteht auch aus Daten. Und wenn wir uns einmal auf diese Sichtweise einlassen, dass Code gleich Daten ist, dann lässt sich Code auch wie Daten übergeben und so von einem Punkt zum anderen übertragen, speichern und später referenzieren. Mit dieser Möglichkeit, Code zu übertragen, lässt sich das Verhalten von Algorithmen leicht anpassen. Beginnen wir mit ein paar Beispielen, bei denen Programmcode übergeben wird, auf den dann später zugegriffen wird:

- Ein Thread führt Programmcode im Hintergrund aus. Der Programmcode, den der Java-Thread ausführen soll, wird in ein Objekt vom Typ `Runnable` verpackt, genau genommen in eine `run()`-Methode gesetzt. Kommt der Thread zum Zuge, ruft er die `run()`-Methode auf.

- Ein `Timer` ist eine `java.util`-Klasse, die zu bestimmen Zeitpunkten Programmcode ausführen kann. Der Objektmethode `scheduleAtFixedRate(…)` wird dabei ein Objekt vom Typ `TimerTask` übergeben, das den Programmcode enthält.
- Zum Sortieren von Daten kann eine eigene Ordnung definiert werden, die dem Sortierer als `Comparator` übergeben werden kann. Der `Comparator` deklariert eine Vergleichsmethode, an die sich der Sortierer wendet, um zwei Objekte in die gewünschte Reihenfolge zu bringen.
- Aktiviert der Benutzer auf der Oberfläche eine Schaltfläche, so führt das zu einer Aktion. Der Programmcode steckt – beim UI-Framework Swing – in einem Objekt vom Typ `ActionListener` und wird an der Schaltfläche `JButton` mit `addActionListener(…)` fest gemacht. Kommt es zu einer Schaltflächenaktivierung, arbeitet das UI-System den Programmcode in der Methode `actionPerformed(…)` des gespeicherten `ActionListener` ab.

Um Programmcode von einer Stelle zur anderen zu bringen, wird in Java immer der gleiche Mechanismus eingesetzt: Eine Klasse implementiert eine (in der Regel nichtstatische) Methode, in der der auszuführende Programmcode steht. Ein Objekt dieser Klasse wird an eine andere Stelle übergeben und der Interessent greift dann über die Methode auf den Programmcode zu. Dass ein Objekt noch mehr als diese eine Implementierung enthalten kann, etwa Variablen, Konstanten, Konstruktoren, ist dafür nicht relevant. Diesen Mechanismus schauen wir uns jetzt in verschiedenen Varianten genauer an.

Innere Klassen als Code-Transporter

Bleiben wir bei dem Beispiel mit den Vergleichen. Angenommen, wir sollen Strings so sortieren, dass Leerraum vorne und hinten bei den Vergleichen ignoriert wird, also " Newton " gleich "Newton" ist. Bei Vorgaben dieser Art muss einem Sortieralgorithmus ein Stückchen Code übergeben werden, damit er die korrekte Reihenfolge herstellen kann. Praktisch sieht das so aus:

```
import java.util.*;

public class CompareTrimmedStrings {
  public static void main( String[] args ) {
    class TrimmingComparator implements Comparator<String> {
      @Override public int compare( String s1, String s2 ) {
        return s1.trim().compareTo( s2.trim() );
      }
    }

    String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };
    Arrays.sort( words, new TrimmingComparator() );
    System.out.println( Arrays.toString( words ) );
  }
}
```


Die Ausgabe ist:

```
[       Adele    , M,  Q,
Skyfall]
```

Der `TrimmingComparator` enthält in der `compare(…)`-Methode den Programmcode für die Vergleichslogik. Ein Exemplar vom `TrimmingComparator` wird aufgebaut und `Arrays.sort(…)` übergeben. Das geht mit weniger Code!

Innere anonyme Klassen als Code-Transporter

Klassen enthalten Programmcode, und Exemplare der Klassen werden an Methoden wie `sort(…)` übergeben, damit der Programmcode dort hinkommt, wo er gebraucht wird. Doch elegant ist das nicht. Für die Beschreibung des Programmcodes ist extra eine eigene Klasse erforderlich. Das ist viel Schreibarbeit, und über eine innere anonyme Klasse lässt sich der Programmcode schon ein wenig verkürzen:

```
String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };
Arrays.sort( words, new Comparator<String>() {
  @Override public int compare( String s1, String s2 ) {
    return s1.trim().compareTo( s2.trim() );
  } } );
System.out.println( Arrays.toString( words ) );
```

Allerdings ist das immer noch aufwändig: Wir müssen eine Methode überschreiben und dann ein Objekt aufbauen. Für Programmautoren ist das lästig, und die JVM hat es mit vielen überflüssigen Klassendeklarationen zu tun.

Abkürzende Schreibweise durch Lambda-Ausdrücke

Ab Java 8 lässt sich Programmcode leichter an eine Methode übergeben, denn es gibt eine kompakte Syntax für die Implementierung von Schnittstellen mit einer Operation. Für unser Beispiel sieht das so aus:

```
String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };
Arrays.sort( words,
            (String s1, String s2) -> { return s1.trim().compareTo(s2.trim()); } );
System.out.println( Arrays.toString( words ) );
```

Der in fett gesetzte Ausdruck nennt sich *Lambda-Ausdruck*. Er ist eine kompakte Art und Weise, Schnittstellen mit genau einer Methode zu implementieren; die Schnittstelle `Comparator` hat genau eine Operation `compare(…)`.

Optisch sind sich ein Lambda-Ausdruck und eine Methodendeklaration ähnlich; was wegfällt sind Modifizierer, der Rückgabetyp, der Methodenname und (mögliche) `throws`-Klauseln.

Methodendeklaration	Lambda-Ausdruck

`public int compare`
`( String s1, String s2 )`
 	
`{ return s1.trim().compareTo( s2.trim() );`
`}`
	
	`( String s1, String s2 )`
	`->`
	`{ return s1.trim().compareTo( s2.trim() );`
	`}`

Tabelle 1.2 Vergleich der Methodendeklaration einer Schnittstelle mit dem Lambda-Ausdruck

Wenn wir uns den Lambda-Ausdruck als Implementierung dieser Schnittstelle anschauen, dann lässt sich dort nichts von `Comparator` oder `compare(…)` ablesen – ein Lambda-Ausdruck repräsentiert mehr oder weniger nur den Java-Code und lässt das, was der Compiler aus dem Kontext herleiten kann, weg.

Alle Lambda-Ausdrücke lassen sich in einer Syntax formulieren, die die folgende allgemeine Form hat:

```
'(' LambdaParameter ')' '->' '{' Anweisungen '}'
```

Lambda-Parameter sind sozusagen die Eingabewerte für die Anweisungen. Die Parameterliste wird so deklariert, wie von Methoden oder Konstruktoren bekannt, allerdings gibt es kein Varargs.

Es gibt syntaktische Abkürzungen, wie wir später sehen werden, doch vorerst bleiben wir bei dieser Schreibweise.

Geschichte

Der Java-Begriff »Lambda-Ausdruck« geht auf das Lambda-Kalkül (engl. *Lambda calculus*, auch geschrieben als λ-calculus) aus den 1930er Jahren zurück und ist eine formale Sprache zur Untersuchung von Funktionen.

1.2.2 Funktionale Schnittstellen und Lambda-Ausdrücke im Detail

In unserem Beispiel haben wir den Lambda-Ausdruck als Argument von `Array.sort(…)` eingesetzt:

```
Arrays.sort( words,
             (String s1, String s2) -> { return s1.trim().compareTo(s2.trim()); } );
```

Wir hätten aber auch den Lambda-Ausdruck explizit einer lokalen Variablen zuweisen können, was deutlich macht, dass der hier eingesetzte Lambda-Ausdruck vom Typ `Comparator` ist:


```
Comparator<String> c = (String s1, String s2) -> {
                           return s1.trim().compareTo( s2.trim() ); }
Arrays.sort( words, c );
```

Funktionale Schnittstellen

Nicht zu jeder Schnittstelle gibt es eine Abkürzung über einen Lambda-Ausdruck, und es gibt eine zentrale Bedingung, wann ein Lambda-Ausdruck verwendet werden kann.

Definition

Schnittstellen, die nur eine Operation (abstrakte Methode) besitzen, heißen *funktionale Schnittstellen*. Ein *Funktions-Deskriptor* beschreibt diese Methode. Eine abstrakte Klasse mit genau einer abstrakten Methode zählt *nicht* als funktionale Schnittstelle.

Lambda-Ausdrücke und funktionale Schnittstellen haben eine ganz besondere Beziehung, denn ein Lambda-Ausdruck ist ein Exemplar einer solchen funktionalen Schnittstelle. Natürlich müssen Typen und Ausnahmen passen. Dass funktionale Schnittstellen genau eine abstrakte Methode vorschreiben, ist eine naheliegende Einschränkung, denn gäbe es mehrere, müsste ein Lambda-Ausdruck ja auch mehrere Implementierungen anbieten oder irgendwie eine Methode bevorzugen und andere ausblenden.

Wenn wir also ein Objekt vom Typ einer funktionalen Schnittstelle aufbauen möchten, können wir folglich zwei Wege einschlagen: Es lässt sich die traditionelle Konstruktion über die Bildung von Klassen wählen, die funktionale Schnittstellen implementieren, und dann mit `new` ein Exemplar bilden, oder es lässt sich mit kompakten Lambda-Ausdrücken arbeiten. Moderne IDEs zeigen uns an, wenn kompakte Lambda-Ausdrücke zum Beispiel statt innerer anonymer Klassen genutzt werden können, und bieten uns mögliche Refactorings an. Lambda-Ausdrücke machen den Code also kompakter und nach kurzer Eingewöhnung auch lesbarer.

Hinweis

Funktionale Schnittstellen müssen auf genau eine zu implementierende Methode hinauslaufen, auch wenn aus Oberschnittstellen mehrere Operationen vorgeschrieben werden, die sich aber durch den Einsatz von Generics auf eine Operation verdichten:

```
interface I<S, T extends CharSequence> {
  void len( S text );
  void len( T text );
}
interface FI extends I<String, String> { }
```

`FI` ist unsere funktionale Schnittstelle mit einer eindeutigen Operation `len(String)`.

Viele funktionale Schnittstellen in der Java-Standardbibliothek

Java 7 bringt schon viele Schnittstellen mit, die in Java 8 als funktionale Schnittstellen gekennzeichnet sind. Darüber hinaus führt Java 8 mit dem Paket `java.util.function` mehr als 40 neue funktionale Schnittstellen ein. Eine kleine Auswahl:

- `interface Runnable { void run(); }`
- `interface Supplier<T> { T get(); }` (Java 8)
- `interface Consumer<T> { void accept(T t); }` (Java 8)
- `interface Comparator<T> { int compare(T o1, T o2); }`
- `interface ActionListener { void actionPerformed(ActionEvent e); }`

Ob die Schnittstelle noch andere Default-Methoden hat – also Schnittstellenmethoden mit vorgegebener Implementierung –, ist egal, wichtig ist nur, dass sie genau eine zu implementierende Operation deklariert.

Typ eines Lambda-Ausdrucks ergibt sich durch Zieltyp

In Java hat jeder Ausdruck einen Typ. `1` und `1*2` haben einen Typ (nämlich `int`), genauso wie `"A" + "B"` (Typ `String`) oder `String.CASE_INSENSITIVE_ORDER` (Typ `Comparator<String>`). Lambda-Ausdrücke haben auch immer einen Typ, denn ein Lambda-Ausdruck ist immer Exemplar einer funktionalen Schnittstelle. Damit steht auch der Typ fest. Allerdings ist es im Vergleich zu Ausdrücken wie `1*2` bei Lambda-Ausdrücken etwas anders gelagert, denn der Typ von Lambda-Ausdrücken ergibt sich ausschließlich aus dem Kontext. Erinnern wir uns an den Aufruf von `sort(…)`:

```
Arrays.sort( words, (String s1, String s2) -> { return … ) } );
```

Dort steht nichts vom Typ `Comparator`, sondern der Compiler erkennt aus dem Typ des zweiten Parameters von `sort(…)`, ob der Lambda-Ausdruck auf `Comparator` passt oder nicht.

Der Typ eines Lambda-Ausdrucks ist also abhängig davon, welche funktionale Schnittstelle er im jeweiligen Kontext gerade realisiert. Der Compiler kann ohne Kenntnis des *Zieltyps* (engl. *target type*) keinen Lambda-Ausdruck aufbauen.

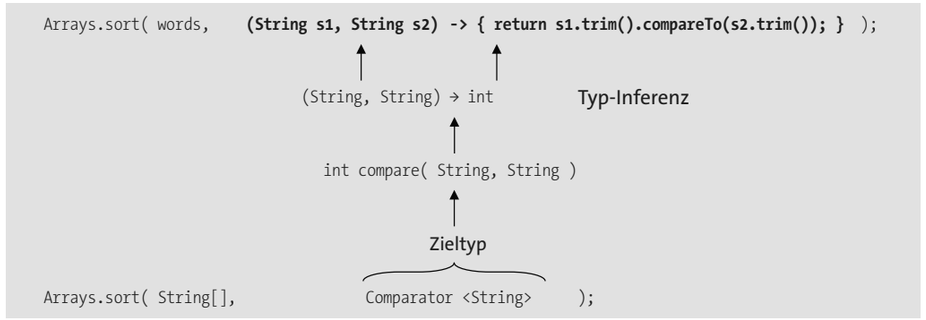

Abbildung 1.1 Typ-Inferenz vom Compiler

Beispiel

`Callable` und `Supplier` sind funktionale Schnittstellen mit Methoden, die keine Parameterlisten deklarieren und eine Referenz zurückgeben; der Code für den Lambda-Ausdruck sieht gleich aus:

```
java.util.concurrent.Callable<String> c = () -> { return "Rückgabe"; };
java.util.function.Supplier<String>   s = () -> { return "Rückgabe"; };
```

Wer bestimmt den Zieltyp?

Gerade weil an dem Lambda-Ausdruck der Typ nicht abzulesen ist, kann er nur dort verwendet werden, wo ausreichend Typinformationen vorhanden sind. Das sind unter anderem die folgenden Stellen:

- Variablendeklarationen: Etwa wie bei `Supplier<String> s = () -> { return "" };`
- Argumente an Methoden oder Konstruktoren: Der Parametertyp gibt alle Typinformationen. Ein Beispiel lieferte `Arrays.sort(…)`.
- Methodenrückgaben: Das könnte aussehen wie `Comparator<String> trimComparator() { return (s1, s2) -> { return … }; }`.
- Bedingungsoperator: Der `?:`-Operator liefert je nach Bedingung einen unterschiedlichen Lambda-Ausdruck. Beispiel: `Supplier<Double> randomNegOrPos = Math.random() > 0.5 ? () -> { return -Math.random(); } : () -> { return Math.random(); };`.

Parametertypen

In der Praxis ist der häufigste Fall, dass die Parametertypen von Methoden den Zieltyp vorgeben. Der Einsatz von Lambda-Ausdrücken ändert ein wenig die Sichtweise auf überladene Methoden. Unser Beispiel mit `() -> { return "Rückgabe"; }` macht das deutlich, denn es »passt« auf den Zieltyp `Callable<String>` genauso wie auf `Supplier<String>`. Nehmen wir an, wir würden zwei überladene Methoden `run(…)` deklarieren:

- `<V> void run( Callable<V> callable ) { }`
- `<V> void run( Supplier<V> callable ) { }`

Spielen wir den Aufruf der Methoden einmal durch:

```
Callable<String> c = () -> { return "Rückgabe"; };
Supplier<String> s = () -> { return "Rückgabe"; };
run( c );
run( s );
// run( () -> { return "Rückgabe"; } ); // ☠ Compilerfehler
run( ((Callable<String>) () -> { return "Rückgabe"; } ) );
```

Rufen wir `run(c)` bzw. `run(s)` auf, ist das kein Problem, denn `c` und `s` sind klar typisiert. Aber `run(…)` mit dem Lambda-Ausdruck aufzurufen funktioniert nicht, denn der Zieltyp (entweder `Callable` oder `Supplier`) ist mehrdeutig; der (Eclipse-)Compiler meldet: »The method run(Callable<Object>) is ambiguous for the type T«. Hier sorgt eine explizite Typanpassung für Abhilfe.

Tipp zum API-Design

Aus Sicht eines API-Designers sind überladene Methoden natürlich schön, aus Sicht des Nutzers sind Typanpassungen aber nicht schön. Um explizite Typanpassungen zu vermeiden, sollte auf überladene Methoden verzichtet werden, wenn diese den Parametertyp einer funktionalen Schnittstelle aufweisen. Stattdessen lassen sich die Methoden unterschiedlich benennen (was bei Konstruktoren natürlich nicht funktioniert).

Wird in unserem Fall die Methode `runCallable(…)` und `runSupplier(…)` genannt, ist keine Typanpassung mehr nötig, und der Compiler kann den Typ herleiten.

Rückgabetypen

Typinferenz spielt bei Lambda-Ausdrücken eine große Rolle – das gilt insbesondere für die Rückgabetypen, die überhaupt nicht in der Deklaration auftauchen und für die es gar keine Syntax gibt; der Compiler »inferred« sie. In unserem Beispiel

```
Comparator<String> c =
  (String s1, String s2) -> { return s1.trim().compareTo( s2.trim() ); };
```

ist `String` als Parametertyp der `Comparator`-Methode ausdrücklich gegeben, der Rückgabetyp `int`, den der Ausdruck `s1.trim().compareTo( s2.trim())` liefert, taucht dagegen nicht auf.

Mitunter muss dem Compiler etwas geholfen werden: Nehmen wir die funktionale Schnittstelle `Supplier<T>`, die eine Methode `T get()` deklariert, für ein Beispiel. Die Zuweisung

```
Supplier<Long> two  = () -> { return 2; }       // ☠ Compilerfehler
```

ist nicht korrekt und führt zum Compilerfehler »incompatible types: bad return type in lambda expression«. `2` ist ein Literal vom Typ `int`, und der Compiler kann es nicht an `Long` anpassen. Wir müssen schreiben

```
Supplier<Long> two  = () -> { return 2L };
```

oder

```
Supplier<Long> two  = () -> { return (long) 2 };
```

Bei Lambda-Ausdrücken gelten keine wirklich neuen Regeln im Vergleich zu Methodenrückgaben, denn auch eine Methodendeklaration wie

```
Long two() { return 2; }                        // ☠ Compilerfehler
```

wird vom Compiler bemängelt. Doch weil Wrapper-Typen durch die Generics bei funktionalen Schnittstellen viel häufiger sind, treten diese Besonderheiten öfter auf als bei Methodendeklarationen.

Sind Lambda-Ausdrücke Objekte?

Ein Lambda-Ausdruck ist ein Exemplar einer funktionalen Schnittstelle und tritt als Objekt auf. Bei Objekten besteht normalerweise zu `java.lang.Object` immer eine natürliche Ist-eine-Art-von-Beziehung. Fehlt aber der Kontext, ist selbst die Ist-eine-Art-von-Beziehung zu `java.lang.Object` gestört und Folgendes nicht korrekt:

```
Object o = () -> {};                    // ☠ Compilerfehler
```

Der Compilerfehler ist: »incompatible types: the target type must be a functional interface«. Nur eine explizite Typanpassung kann den Fehler korrigieren und dem Compiler den Zieltyp vorgeben:

```
Object r = (Runnable) () -> {};
```

Lambda-Ausdrücke haben also keinen eigenen Typ an sich, und für das Typsystem von Java ändert sich im Prinzip nichts. Möglicherweise ändert sich das in späteren Java-Versionen.

Hinweis

Dass Lambda-Ausdrücke Objekte sind, ist eine Eigenschaft, die nicht überstrapaziert werden sollte. So sind die üblichen `Object`-Methoden `equals(Object)`, `hashCode()`, `getClass()`, `toString()` und die zur Thread-Kontrolle ohne besondere Bedeutung. Es sollte auch nie ein Szenario geben, in dem Lambda-Ausdrücke mit `==` verglichen werden müssen, denn das Ergebnis ist laut Spezifikation undefiniert. Echte Objekte haben eine Identität, einen Identity-Hashcode, lassen sich vergleichen und mit `instanceof` testen, können mit einem synchronisierten Block abgesichert werden; all dies gilt für Lambda-Ausdrücke nicht. Im Grunde charakterisiert der Begriff »Lambda-Ausdruck« schon sehr gut, was wir nie vergessen sollten: Es handelt sich um einen Ausdruck, also etwas, das ausgewertet wird und ein Ergebnis produziert.

Annotation @FunctionalInterface

Jede Schnittstelle mit genau einer abstrakten Methode eignet sich als funktionale Schnittstelle und damit für einen Lambda-Ausdruck. Jedoch soll nicht jede Schnittstelle in der API, die im Moment nur eine abstrakte Methode deklariert, auch für Lambda-Ausdrücke verwendet werden. So kann zum Beispiel eine Weiterentwicklung der Schnittstelle mit mehreren (abstrakten) Methoden geplant sein, aber zurzeit ist nur eine abstrakte Methode vorhanden. Der Compiler kann nicht wissen, ob sich eine Schnittstelle vielleicht weiterentwickelt. Um kenntlich zu machen, dass ein `interface` als funktionale Schnittstelle gedacht ist, existiert der Annotationstyp `FunctionalInterface` im `java.lang`-Paket. Diese markiert, dass es bei genau einer abstrakten Methode und damit bei einer funktionalen Schnittstelle bleiben soll.

Beispiel

Eine eigene funktionale Schnittstelle sollte immer als `FunctionalInterface` markiert werden:

```
@FunctionalInterface
public interface MyFunctionalInterface {
  void foo();
}
```

Der Compiler prüft, ob die Schnittstelle mit einer solchen Annotation tatsächlich nur exakt eine abstrakte Methode enthält, und löst einen Fehler aus, wenn dem nicht so ist. Aus Kompatibilitätsgründen erzwingt der Compiler diese Annotation bei funktionalen Schnittstellen allerdings nicht; das ermöglicht es, dass innere Klassen, die herkömmliche Schnittstellen mit einer Methode implementieren, einfach in Lambda-Ausdrücke umgeschrieben werden können. Die Annotation ist also keine Voraussetzung für die Nutzung der Schnittstelle in einem Lambda-Ausdruck und dient bisher nur der Dokumentation. In der Java SE sind aber alle zentralen funktionalen Schnittstellen so ausgezeichnet.

Syntax für Lambda-Ausdrücke

Lambda-Ausdrücke haben wie Methoden mögliche Parameter- und Rückgabewerte. Die Java-Grammatik für die Schreibweise von Lambda-Ausdrücken sieht ein paar nützliche syntaktische Abkürzungen vor.

Ausführliche Schreibweise

Lambda-Ausdrücke lassen sich auf unterschiedliche Art und Weise schreiben, da es für diverse Konstruktionen Abkürzungen gibt. Eine Form, die jedoch immer gilt ist:

```
'(' LambdaParameter ')' '->' '{' Anweisungen '}'
```

Der Lambda-Parameter besteht (voll ausgeschrieben) wie ein Methodenparameter aus a) dem Typ, b) dem Namen und c) optionalen Modifizierern.

Der Parametername öffnet einen neuen Gültigkeitsbereich für eine Variable, wobei der Parametername *keine* anderen Namen von lokalen Variablen überlagern darf. Hier verhält sich die Lambda-Parametervariable wie eine neue Variable aus einem inneren Block und nicht wie eine Variable aus einer inneren Klasse, wo die Sichtbarkeit anders ist.

Beispiel

Folgendes ergibt einen Compilerfehler im Lambda-Ausdruck, weil `var` schon deklariert ist; die Parametervariable vom Lambda-Ausdruck muss also »frisch« sein:

```
String var = "";
var.chars().forEach( var -> { System.out.println( var ); } );  // ☠ Compilerfehler
```


Abkürzung 1: Typinferenz

Der Java-Compiler kann viele Typen aus dem Kontext ablesen, was Typ-Inferenz genannt wird. Wir kennen so etwas vom Diamant-Operator, wenn wir etwa schreiben:

```
List<String> list = new ArrayList<>()
```

Sind für den Compiler genug Typ-Informationen verfügbar, dann erlaubt der Compiler bei Lambda-Ausdrücken eine Abkürzung. Bei

```
Comparator<String> c =
  (String s1, String s2) -> { return s1.trim().compareTo( s2.trim() ); };
```

ist Typ-Inferenz einfach (`Comparator<String>` sagt alles aus), daher funktioniert die folgende Abkürzung:

```
Comparator<String> c = (s1, s2) -> { return s1.trim().compareTo( s2.trim() ); };
```

Die Parameterliste enthält also entweder explizit deklarierte Parametertypen oder implizite Inferred-Typen. Eine Mischung ist nicht erlaubt, der Compiler blockt so etwas wie `(String s1, s2)` oder `(s1, String s2)` mit einem Fehler ab.

Wenn der Compiler die Typen ablesen kann, sind die Parametertypen optional. Aber Typ-Inferenz ist nicht immer möglich, weshalb die Abkürzung nicht immer möglich ist. Außerdem hilft die explizite Schreibweise auch der Lesbarkeit: Kurze Ausdrücke sind nicht unbedingt die verständlichsten.

Hinweis

Der Compiler liest aus den Typen ab, ob alle Eigenschaften vorhanden sind. Die Typen sind dabei entweder explizit oder implizit gegeben.

```
Comparator<String> sc = (a, b) -> { return Integer.compare( a.length(), b.length() ); };
Comparator<BitSet> bc = (a, b) -> { return Integer.compare( a.length(), b.length() ); };
```

Die Klassen `String` und `BitSet` besitzen beide die Methode `length()`, daher ist der Lambda-Ausdruck korrekt. Der gleiche Lambda-Code lässt sich für zwei völlig verschiedene Klassen einsetzen, die überhaupt keine Gemeinsamkeiten haben, nur dass sie zufällig beide eine Methode namens `length()` besitzen.

Abkürzung 2: Lambda-Rumpf ist entweder einzelner Ausdruck oder Block

Besteht der Rumpf eines Lambda-Ausdrucks nur aus einem einzelnen Ausdruck, kann eine verkürzte Schreibweise die Blockklammern und das Semikolon einsparen. Statt

```
( LambdaParameter ) -> { return Ausdruck; }
```

heißt es dann

```
( LambdaParameter ) -> Ausdruck
```

Lambda-Ausdrücke mit einer `return`-Anweisung im Rumpf kommen häufig vor, da dies den typischen Funktionen entspricht. Somit ist es eine willkommene Verkürzung, wenn die abgekürzte Syntax für Lambda-Ausdrücke lediglich den Ausdruck fordert, der dann die Rückgabe bildet. Hier sind drei Beispiele:

Lange Schreibweise	Abkürzung
`(s1, s2) ->` `{ return s1.trim().compareTo( s2.trim() ); }`	`(s1, s2) ->` `s1.trim().compareTo( s2.trim() )`
`(a, b) -> { return a + b; }`	`(a, b) -> a + b`
`() -> { System.out.println(); }`	`() -> System.out.println()`

Tabelle 1.3 Ausführliche und abgekürzte Schreibweise

Ausdrücke können in Java auch zu `void` ausgewertet werden, sodass ohne Probleme ein Aufruf wie `System.out.println()` in der kompakten Schreibweise ohne Block gesetzt werden kann. Das heißt, wenn Lambda-Ausdrücke mit der kurzen Ausdruckssyntax eingesetzt werden, können diese Ausdrücke etwas zurückgeben, müssen aber nicht.

[+]

Hinweis

Die Schreibweise mit den geschweiften Klammern und den Rückgabe-Ausdrücken kann nicht gemischt werden. Entweder gibt es einen Block geschweifter Klammern und `return` oder keine Klammern und kein `return`-Schlüsselwort. Fehler ergeben also diese falschen Mischungen:

```
Comparator<String> c;
c = (s1, s2) -> { s1.trim().compareTo( s2.trim() ) };    // ☠ Compilerfehler (1)
c = (s1, s2) -> return s1.trim().compareTo( s2.trim() ); // ☠ Compilerfehler (2)
```

Würden wir in `(1)` ein explizites `return` nutzen wäre alles in Ordnung, würde bei `(2)` das `return` wegfallen, wäre die Zeile auch compilierbar.

Ob Lambda-Ausdrücke eine Rückgabe haben, drücken zwei Begriffe aus:

- `void`-kompatibel: Der Lambda-Rumpf gibt kein Ergebnis zurück, entweder weil der Block kein `return` enthält oder ein `return` ohne Rückgabe oder weil ein `void`-Ausdruck in der verkürzten Schreibweise eingesetzt wird. Der Lambda-Ausdruck `() -> System.out.println()` ist also `void`-kompatibel, genauso wie `() -> {}`.
- Wert-kompatibel: Der Rumpf beendet den Lambda-Ausdruck mit einer `return`-Anweisung, die einen Wert zurückgibt, oder besteht aus der kompakten Schreibenweise mit einer Rückgabe ungleich `void`.

Eine Mischung aus `void`- und Wert-kompatibel ist nicht erlaubt und führt wie bei Methoden zu einem Compilerfehler.[8]

8 Wohl aber gibt es wie bei `{ throw new RuntimeException(); }` Ausnahmen, bei denen Lambda-Ausdrücke beides sind.

Abkürzung 3: Einzelner Identifizierer statt Parameterliste und Klammern

Besteht die Parameterliste

1. nur aus einem einzelnen Identifizierer
2. und ist der Typ durch Typ-Inferenz klar,

können die runden Klammern wegfallen.

Lange Schreibweise	Typen inferred	Vollständig abgekürzt
`(String s) -> s.length()`	`(s) -> s.length()`	`s -> s.length()`
`(int i) -> Math.abs( i )`	`(i) -> Math.abs( i )`	`i -> Math.abs( i )`

Tabelle 1.4 Unterschiedlicher Grad von Abkürzungen

Kommen alle Abkürzungen zusammen, lässt sich etwa die Hälfte an Code einsparen. Aus `(int i) -> { return Math.abs( i ); }` wird einfach `i -> Math.abs( i )`.

Syntax-Hinweis

Nur bei genau einem Lambda-Parameter können die Klammern weggelassen werden, da es sonst Mehrdeutigkeiten gibt, für die es wieder komplexe Regeln zur Auflösung geben müsste. Heißt es etwa `foo( k, v -> { … } )`, ist unklar, ob `foo` zwei Parameter deklariert. Ist das zweite Argument ein Lambda-Ausdruck, oder handelt es sich um nur genau einen Parameter, wobei dann ein Lambda-Ausdruck übergeben wird, der selbst zwei Parameter deklariert? Um Problemen wie diesen aus dem Weg zu gehen, können Entwickler auf den ersten Blick sehen, dass `foo( k, v -> { … } )` eindeutig für Parameter steht, und `foo( (k, v) -> { … } )` nur einen Parameter besitzt.

Unbenutzte Parameter in Lambda-Ausdrücken

Es kommt vor, dass ein Lambda-Ausdruck eine funktionale Schnittstelle implementiert, aber nicht jeder Parameter von Interesse ist. Als Beispiel schauen wir uns an:

```
interface Consumer<A> { void apply( A a ); }
```

Ein Konsument, der das Argument in Hochkommata ausgibt, sieht so aus:

```
Consumer<String> printQuoted = s -> System.out.printf( "'%s'", s );
printQuoted.accept( "Chris" );  // 'Chris'
```

Was ist nun, wenn ein Konsument auf das Argument gar nicht zugreifen möchte, weil zum Beispiel die aktuelle Zeit ausgegeben wird?

```
Consumer<String> printNow =
  s -> System.out.println( System.currentTimeMillis() );
```

Die Variable `s` in der Lambda-Parameterliste ist ungenutzt und wird vom Compiler auch als »unused« bemängelt. Daher erlaubt eine spezielle Schreibweise, den Variablennamen wegzulassen und nur den Typ anzugeben:

```
Consumer<String> printNow =
  String /* bzw. (String)*/ -> System.out.println( System.currentTimeMillis() );
```

Der Typ selbst darf nicht entfallen, denn `() -> …` passt nicht auf die funktionale Schnittstelle `Consumer`, die immer einen Parameter deklariert.

Hinweis

Nur bei einem Lambda-Parameter ist diese Form der Abkürzung erlaubt. Folgendes führt zu einem Compilerfehler:

```
BiFunction<Double, Double, Double> bifunc = (Double a, Double /* ☠ */)
  -> Math.signum( a );
```

Die Umgebung der Lambda-Ausdrücke und Variablenzugriffe

Ein Lambda-Ausdruck »sieht« seine Umgebung genauso wie der Code, der vor oder nach dem Lambda-Ausdruck steht. Insbesondere hat ein Lambda-Ausdruck vollständigen Zugriff auf alle Eigenschaften der Klasse, genauso wie auch der einschließende äußere Block sie hat. Es gibt keinen besonderen Namensraum, sondern nur neue und vielleicht überdeckte Variablen durch die Parameter. Das ist einer der grundlegenden Unterschiede zwischen Lambda-Ausdrücken und inneren Klassen. Somit ist auch die Bedeutung von `this` und `super` bei Lambda-Ausdrücken und inneren Klassen unterschiedlich.

Zugriff auf finale, lokale Variablen/Parametervariablen

Lambda-Ausdrücke können problemlos auf Objektvariablen und Klassenvariablen lesend und schreibend zugreifen. Auch auf lokale Variablen und Parameter hat ein Lambda-Ausdruck Zugriff. Doch greift ein Lambda-Ausdruck auf lokale Variablen bzw. Parametervariablen zu, müssen diese final sein. Dass eine Variable final ist, muss nicht extra mit einem Modifizierer geschrieben werden, aber sie muss *effektiv final* (engl. *effectively final*) sein. Effektiv final ist eine Variable, wenn sie nach der Initialisierung nicht mehr beschrieben wird.

Ein Beispiel: Der Benutzer soll über eine Eingabe die Möglichkeit bekommen, zu bestimmen, ob String-Vergleiche mit unserem trimmenden `Comparator` unabhängig von der Groß-/Kleinschreibung stattfinden sollen:

```
public class CompareIgnoreCase {
  public static void main( String[] args ) {
    /*final*/ boolean compareIgnoreCase = new Scanner( System.in ).nextBoolean();
    Comparator<String> c = (s1, s2) -> compareIgnoreCase ?
                                       s1.trim().compareToIgnoreCase( s2.trim() ) :
                                       s1.trim().compareTo( s2.trim() );
    String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };
    Arrays.sort( words, c );
    System.out.println( Arrays.toString( words ) );
  }
}
```


Ob `compareIgnoreCase` von uns `final` gesetzt wird oder nicht, ist egal, denn die Variable wird hier effektiv final verwendet. Natürlich kann es nicht schaden, `final` als Modifizierer immer davorzusetzen, um dem Leser des Codes diese Tatsache bewusst zu machen.

Neu eingeschobene Lambda-Ausdrücke, die auf lokale Variablen bzw. Parametervariablen zugreifen, können also im Nachhinein zu Compilerfehlern führen. Folgendes Segment ist ohne Lambda-Ausdruck korrekt:

```
/*1*/ boolean compareIgnoreCase = new Scanner( System.in ).nextBoolean();
/*2*/ …
/*3*/ compareIgnoreCase = true;
```

Schiebt sich zwischen Zeile 1 und 3 nachträglich ein Lambda-Ausdruck, der auf `compareIgnoreCase` zugreift, gibt es anschließend einen Compilerfehler. Allerdings liegt der Fehler nicht in Zeile 3, sondern beim Lambda-Ausdruck. Denn die Variable `compareIgnoreCase` ist nach der Änderung nicht mehr effektiv final, was sie aber sein müsste, um in dem Lambda-Ausdruck verwendet zu werden.

Tipp

Lambda-Ausdrücke verhalten sich genauso wie innere anonyme Klassen, die auch nur auf finale Variablen zugreifen können. Mit Behältern wie einem Feld oder den speziellen `AtomicXXX`-Klassen aus dem `java.util.concurrent.atomic`-Paket lässt sich das Problem im Prinzip lösen. Denn greift ein Lambda-Ausdruck etwa auf das Feld `boolean[] compareIgnoreCase = new boolean[1];` zu, so ist die Variable `compareIgnoreCase` selbst final, aber `compareIgnoreCase[0] = true;` ist erlaubt, denn es ist ein Schreibzugriff auf das Feld, nicht auf die Variable `compareIgnoreCase`. Je nach Code besteht jedoch die Gefahr, dass Lambda-Ausdrücke parallel ausgeführt werden. Wird etwa ein Lambda-Ausdruck mit Veränderung auf diesem Feldinhalt parallel ausgeführt, so ist der Zugriff nicht synchronisiert, und das Ergebnis kann »kaputt« sein, denn paralleler Zugriff auf Variablen muss immer koordiniert vorgenommen werden.

Namensräume

Deklariert eine innere anonyme Klasse Variablen innerhalb der Methode, so sind diese immer »neu«, das heißt, die neuen Variablen überlagern vorhandene lokale Variablen aus dem äußeren Kontext. Die Variable `compareIgnoreCase` kann im Rumpf von `compare(…)` zum Beispiel problemlos neu deklariert werden:

```
boolean compareIgnoreCase = true;
Comparator<String> c = new Comparator<String>() {
  @Override public int compare( String s1, String s2 ) {
   boolean compareIgnoreCase = false;        // völlig ok
   return …
  }
};
```

In einem Lambda-Ausdruck ist das nicht möglich, und Folgendes führt zu einer Fehlermeldung des Compilers: »variable compareIgnoreCase is already defined«.

```
boolean compareIgnoreCase = true;
Comparator<String> c = (s1, s2) -> {
  boolean compareIgnoreCase = false;  // ☠ Compilerfehler
  return …
}
```

this-Referenz

Ein Lambda-Ausdruck unterscheidet sich von einer inneren (anonymen) Klasse auch in dem, worauf die `this`-Referenz verweist:

- Beim Lambda-Ausdruck zeigt `this` immer auf das Objekt, in dem der Lambda-Ausdruck eingebettet ist.
- Bei einer inneren Klasse referenziert `this` die innere Klasse, und die ist ein komplett neuer Typ.

Folgendes Beispiel macht das deutlich:

Listing 1.4 InnerVsLambdaThis.java

```
class InnerVsLambdaThis {

  InnerVsLambdaThis() {
    Runnable lambdaRun = () -> System.out.println( this.getClass().getName() );
    Runnable innerRun  = new Runnable() {
      @Override public void run() { System.out.println( this.getClass().getName()); }
    };

    lambdaRun.run();      // InnerVsLambdaThis
    innerRun.run();       // InnerVsLambdaThis$1
  }

  public static void main( String[] args ) {
    new InnerVsLambdaThis();
  }
}
```


Als Erstes nutzen wir `this` in einen Lambda-Ausdruck im Konstruktor der Klasse `InnerVsLambdaThis`. Damit bezieht sich `this` auf jedes gebaute `InnerVsLambdaThis`-Objekt. Bei der inneren Klasse referenziert `this` ein anderes Exemplar, und zwar vom Typ `Runnable`. Da es bei anonymen Kassen keinen Namen hat, trägt es lediglich die Kennung `InnerVsLambdaThis$1`.

Rekursive Lambda-Ausdrücke

Lambda-Ausdrücke können auf sich selbst verweisen. Da aber ein `this` zur Selbstreferenz nicht funktioniert, ist ein kleiner Umweg nötig. Erst muss eine Objekt- oder eine Klassenvariable deklariert werden, dann muss dieser Variablen ein Lambda-Ausdruck zugewiesen werden, und dann kann der Lambda-Ausdruck auf diese Variable zugreifen und einen rekursiven Aufruf starten. Für den Klassiker der Fakultät sieht das so aus:

```
public class RecursiveFactLambda {
  public static IntFunction<Integer> fact = n -> (n == 0) ? 1 : n * fact.apply(n-1);
  public static void main( String[] args ) {
    System.out.println( fact.apply( 5 ) );   // 120
  }
}
```

`IntFunction` ist eine funktionale Schnittstelle aus dem Paket `java.util.function` mit einer Operation `T apply(int i)`. `T` ist ein generischer Rückgabetyp, den wir hier mit `Integer` belegt haben.

`fact` hätte genauso gut als normale Methode deklariert werden können. Großartige Vorteile bietet die Schreibweise mit Lambda-Ausdrücken hier nicht. Zumal jetzt auch der Begriff *anonyme Methode* nicht mehr so richtig passt, da der Lambda-Ausdruck ja doch einen Namen hat, nämlich `fact`. Und weil der Lambda-Ausdruck einer Variablen zugewiesen wurde, kann er in dieser Form natürlich auch nicht mehr als Implementierung an eine Methode oder einen Konstruktor übergeben werden, sondern nur als Methoden-/Konstruktor-Referenz, dazu später mehr.

Ausnahmen in Lambda-Ausdrücken

Lambda-Ausdrücke sind Implementierungen von funktionalen Schnittstellen, und bisher haben wir noch nicht die Frage betrachtet, was passiert, wenn der Codeblock vom Lambda-Ausdruck eine Ausnahme auslöst, und wer diese auffangen muss.

Ausnahmen im Codeblock eines Lambda-Ausdrucks

In `java.util.function` gibt es eine funktionale Schnittstelle `Predicate`, deren Deklaration im Kern wie folgt ist:

```
public interface Predicate<T> { boolean test( T t ); }
```

Ein `Predicate` führt einen Test durch und liefert wahr oder falsch als Ergebnis. Ein Lambda-Ausdruck kann diese Schnittstelle nun implementieren. Nehmen wir an, wir wollten testen, ob eine Datei die Länge 0 hat, um etwa Dateileichen zu finden. In einer ersten Idee greifen wir auf die existierende `Files`-Klasse zurück, die `size(…)`anbietet:

```
Predicate<Path> isEmptyFile = path -> Files.size( path ) == 0;  // ☠ Compilerfehler
```

Problem dabei ist, dass `Files.size(…)` eine `IOException` auslöst, die behandelt werden muss, und zwar *nicht* vom Block, in dem der Lambda-Ausdruck als Ganzes steht, sondern vom Code im Lambda-Ausdruck selbst. Das schreibt der Compiler so vor. Folgendes ist also keine Lösung:

```
try {
  Predicate<Path> isEmptyFile = path -> Files.size( path ) == 0; // ☠
} catch ( IOException e ) { … }
```

sondern nur:

```
Predicate<Path> isEmptyFile = path -> {
  try {
    return Files.size( path ) == 0;
  } catch ( IOException e ) { return false; }
};
```

Die Eigenschaft, die Java fehlt, nennt sich *Exception-Transparenz*, und hier ist deutlich der Unterschied zwischen geprüften und ungeprüften Ausnahmen zu sehen. Bei der Exception-Transparenz wäre keine Ausnahmebehandlung im Lambda-Ausdruck nötig und an einer übergeordneten Stelle möglich. Doch da diese Möglichkeit in Java fehlt, bleibt uns nur übrig, geprüfte Ausnahmen im Lambda-Ausdrücken direkt zu behandeln.

Funktionale Schnittstellen mit throws-Klausel

Ungeprüfte Ausnahmen können immer auftreten und führen (nicht abgefangen) wie üblich zum Abbruch des Threads. Eine `throws`-Klausel an den Methoden/Konstruktoren ist dafür nicht nötig. Doch können funktionale Schnittstellen eine `throws`-Klausel mit geprüften Ausnahmen deklarieren, und die Implementierung einer funktionalen Schnittstelle kann logischerweise geprüfte Ausnahmen auslösen.

Eine Deklaration wie `Callable` aus dem Paket `java.util.concurrent` macht das deutlich. (`Callable` trägt kein `@FunctionalInterface`):

```
public interface Callable<V> {
  V call() throws Exception;
}
```

Das könnte durch folgenden Lambda-Ausdruck realisiert werden:

```
Callable<Integer> randomDice = () -> (int)(Math.random() * 6) + 1;
```

Der Aufruf von `call()` auf einem `randomDice` muss mit einer Ausnahmebehandlung einhergehen, da `call()` eine `Exception` auslöst, etwa so:

```
try {
  System.out.println( randomDice.call() );
  System.out.println( randomDice.call() );
}
catch ( Exception e ) { … }
```


Dass der Aufrufer die Ausnahme behandeln muss, ist klar. Die Deklaration des Lambda-Ausdrucks enthält keinen Hinweis auf die Ausnahme, das ist ein Unterschied zum vorangegangenen Abschnitt.

Design-Tipp

Ausnahmen in Methoden funktionaler Schnittstellen schränken den Nutzen stark ein, und daher löst keine der funktionalen Schnittstellen aus etwa `java.util.function` eine geprüfte Ausnahme aus. Der Grund ist einfach, denn jeder Methodenaufrufer müsste sonst entweder die Ausnahme weiterleiten oder behandeln.[9]

Um die Einschränkungen und Probleme mit einer `throws`-Klausel noch etwas deutlicher zu machen, stellen wir uns vor, dass die funktionale Schnittstelle `Predicate` ein `throws Exception` (vom Sinn der Typs `Exception` an sich einmal abgesehen) enthält:

```
interface Predicate<T> { boolean test( T t ) throws Exception; } // Was wäre wenn?
```

Die Konsequenz wäre, dass jeder Aufrufer von `test(…)` nun seinerseits die `Exception` in die Hände bekäme und sie auffangen oder weiterleiten müsste. Leitet der `test(…)`-Aufrufer mit `throws Exception` die Ausnahme weiter nach oben, bekommen wir plötzlich an allen Stellen ein `throws Exception` in die Methodensignatur, was auf keinen Fall gewünscht ist. So enthält jetzt etwa `ArrayList` eine Deklaration von `removeIf(Predicate filter)`; hier müsste sich dann `removeIf(…)` – was letztendlich `filter.test(…)` aufruft – mit der Testausnahme rumärgern, und `removeIf(Predicate filter) throws Exception` ist keine gute Sache.

Von geprüft nach ungeprüft

Geprüfte Ausnahmen sind in Lamba-Ausdrücken nicht schön. Eine Lösung ist, Code, der geprüfte Ausnahmen auslöst, zu verpacken und die geprüfte Ausnahme in einer ungeprüften zu manteln. Das kann etwa so aussehen:

Listing 1.5 PredicateWithException.java

```
public class PredicateWithException {

  @FunctionalInterface
  public interface ExceptionalPredicate<T, E extends Exception> {
    boolean test( T t ) throws E;
  }

  public static <T> Predicate<T> asUncheckedPredicate(
```

9 Von `Callable` gibt es zwar Nutzer, die mit Nebenläufigkeit (daher das Paket `java.util.concurrent`) in Zusammenhang stehen, aber keine weiteren Verwendungen in der Java-Bibliothek, von zwei Beispielen aus `javax.tools` abgesehen. Mit `java.util.function.Supplier` existiert eine entsprechende Alternative ohne `throws`-Klausel.

```
    ExceptionalPredicate<T, Exception> predicate ) {
    return t -> {
      try {
        return predicate.test( t );
      }
      catch ( Exception e ) {
        throw new RuntimeException( e.getMessage(), e );
      }
    };
  }

  public static void main( String[] args )  {
    Predicate<Path> isEmptyFile = asUncheckedPredicate( path -> Files.size( path ) == 0 );
    System.out.println( isEmptyFile.test( Paths.get( "c:/" ) ) );
  }
}
```

Die Schnittstelle `ExceptionalPredicate` ist ein Prädikat mit optionaler Ausnahme. In der eigenen Hilfsmethode `asUncheckedPredicate(ExceptionalPredicate)` nehmen wir so ein `ExceptionalPredicate` an und packen es in ein `Predicate`, was die Methode zurückgibt. Geprüfte Ausnahmen werden in eine ungeprüfte Ausnahme vom Typ `RuntimeException` gesetzt. Somit muss `Predicate` keine geprüfte Ausnahme weiterleiten, was es ja laut Deklaration auch nicht kann.

Die Java-Bibliothek selbst bringt keine Ummantelungen dieser Art mit. Es gibt nur eine interne Methode, die etwas Vergleichbares tut:

Listing 1.6 java.nio.file.Files.java, asUncheckedRunnable(...)

```
/**
 * Convert a Closeable to a Runnable by converting checked IOException
 * to UncheckedIOException
 */
private static Runnable asUncheckedRunnable( Closeable c ) {
  return () -> {
    try {
      c.close();
    }
    catch ( IOException e ) {
      throw new UncheckedIOException( e );
    }
  };
}
```


Hier kommt die Klasse `UncheckedIOException` zum Einsatz. Diese ist eine ungeprüfte Ausnahme, die als Wrapper-Klasse für Ein-/Ausgabefehler genutzt wird. Wir finden die `UncheckedIOException` etwa bei `lines()` von `BufferedReader` bzw. `Files`, die einen `Stream<String>` mit Zeilen liefert – geprüfte Ausnahmen sind hier nur im Weg.

Klassen mit einer abstrakten Methode als funktionale Schnittstelle? *

Als die Entwickler der Sprache Java die Lambda-Ausdrücke diskutierten, stand auch die Frage im Raum, ob abstrakte Klassen, die nur über eine abstrakte Methode verfügen, ebenfalls für Lambda-Ausdrücke genutzt werden können.[10] Sie entschieden sich dagegen, unter anderem deswegen, weil bei der Implementierung von Schnittstellen die JVM weitreichende Optimierungen vornehmen kann. Und bei Klassen wird das schwierig. Das liegt auch daran, dass ein Konstruktor umfangreiche Initialisierungen mit Seiteneffekten vornimmt (die Konstruktoren aller Oberklassen nicht zu vergessen) sowie Ausnahmen auslösen könnte. Gewünscht ist aber nur die Ausführung einer Implementierung der funktionalen Schnittstelle und kein anderer Code.

Es gibt nun im JDK einige abstrakte Klassen, die genau eine abstrakte Methode vorschreiben, etwa `java.util.TimerTask`. Solche Klassen können nicht über einen Lambda-Ausdruck realisiert werden; hier müssen Entwickler weiterhin zu Klassenimplementierungen greifen, und die kürzeste Lösung ist eine innere anonyme Klasse. Eigene Hilfsklassen können natürlich den Code etwas abkürzen, aber eben nur mithilfe einer eigenen Implementierung.

Wer abstrakte Methoden mit Lambda-Ausdrücken implementieren möchte, kann mit Hilfsklassen arbeiten. Denn wenn eine Hilfsklasse funktionale Schnittstellen einsetzt, so können Lambda-Ausdrücke wieder ins Spiel kommen, indem die Implementierung der abstrakten Methode an den Lambda-Ausdruck weiterleitet. Nehmen wir das Beispiel für `TimerTask` und gehen zwei unterschiedliche Strategien der Implementierung durch. Mit Delegation sieht das so aus:

```
import java.util.*;

class TimerTaskLambda {

  public static TimerTask createTimerTask( Runnable runnable ) {
    return new TimerTask() {
        @Override public void run() { runnable.run(); }
    };
  }

  public static void main( String[] args ) {
    new Timer().schedule( createTimerTask( () -> System.out.println("Hi") ), 500 );
  }
}
```

10 Früher wurde hier die Abkürzung *SAM (Single Abstract Method)* genutzt.

Mit Vererbung erhalten wir:

```
public class LambdaTimerTask extends TimerTask {

  private final Runnable runnable;

  public LambdaTimerTask( Runnable runnable ) {
    this.runnable = runnable;
  }

  @Override public void run() { runnable.run();  }
}
```

Der Aufruf erfolgt dann statt über `createTimerTask(…)` mit dem Konstruktor:

```
new Timer().schedule( new LambdaTimerTask( () -> System.out.println("Hi") ), 500 );
```

1.2.3 Methoden-Referenz

Je größer Softwaresysteme werden, desto wichtiger werden Dinge wie Klarheit, Wiederverwendbarkeit und Dokumentation. Wir haben für unseren String-Comparator eine Implementierung geschrieben, anfangs über eine innere Klasse, später über einen Lambda-Ausdruck. In jedem Fall haben wir Code geschrieben. Doch was wäre, wenn eine Utility-Klasse schon eine Implementierung mitbringen würde? Dann könnte der Lambda-Ausdruck natürlich an die vorhandene Implementierung delegieren, und wir sparen Code.

Schauen wir uns das mal an einem Beispiel an:

```
class StringUtils {
  public static int compareTrimmed( String s1, String s2 ) {
    return s1.trim().compareTo( s2.trim() );
  }
}

public class CompareIgnoreCase {
  public static void main( String[] args ) {
    String[] words = { "A", "B", "a" };
      Arrays.sort( words, (String s1, String s2) -> StringUtils.compareTrimmed(s1, s2) );
      System.out.println( Arrays.toString( words ) );
  }
}
```


Auffällig ist hier, dass die referenzierte Methode `compareTrimmed(String,String)` von den Parametertypen und vom Rückgabetyp genau auf die `compare(…)`-Methode eines `Comparator` passt. Für genau solche Fälle gibt es eine weitere syntaktische Verkürzung, sodass im Code kein Lambda-Ausdruck, sondern nur noch ein Methodenverweis notwendig ist.

Definition

Eine *Methoden-Referenz* ist ein Verweis auf eine Methode, ohne diese jedoch aufzurufen. Syntaktisch trennen zwei Doppelpunkte den Klassenamen bzw. die Referenz auf der linken Seite von dem Methodennamen auf der rechten.

Die Zeile

```
Arrays.sort( words, (String s1, String s2) -> StringUtils.compareTrimmed(s1, s2) );
```

lässt sich mit einer Methoden-Referenz abkürzen zu:

```
Arrays.sort( words, StringUtils::compareTrimmed );
```

Die Sortiermethode erwartet vom `Comparator` eine Methode, die zwei Strings annimmt und eine Ganzzahl zurückgibt. Der Name der Klasse und der Name der Methode sind unerheblich, weshalb an dieser Stelle eine Methoden-Referenz eingesetzt werden kann.

Eine Methoden-Referenz ist wie ein Lambda-Ausdruck ein Exemplar einer funktionalen Schnittstelle, jedoch für eine existierende Methode einer bekannten Klasse. Wie üblich bestimmt der Kontext, von welchem Typ genau der Ausdruck ist.

Hinweis

Gleicher Code für eine Methoden-Referenz kann zu komplett unterschiedlichen Typen führen – der Kontext macht den Unterschied:

```
Comparator<String>                  c1 = StringUtils::compareTrimmed;
BiFunction<String, String, Integer> c2 = StringUtils::compareTrimmed;
```

Varianten von Methoden-Referenzen

Im Beispiel ist die Methode `compareTrimmed(…)` statisch, und links vom Doppelpunkt steht der Name eines Typs. Allerdings kann beim Einsatz eines Typnamens die Methode auch nichtstatisch sein, `String::length` ist so ein Beispiel. Das wäre eine Funktion, die ein `String` auf ein `int` abbildet, in Code:

```
Function<String, Integer> len = String::length;
```

Links von den zwei Doppelpunkten kann auch eine Referenz stehen, was dann immer eine Objektmethode referenziert.

Beispiel

Während `String::length` eine Funktion ist, wäre `string::length` ein `Supplier`, unter der Annahme, dass `string` eine Referenzvariable ist:

```
String string = "Goll";
Supplier<Integer> len = string::length;
System.out.println( len.get() );     // 4
```

`System.out` ist eine Referenz, und eine Methode wie `println(…)` kann an einen `Consumer` gebunden werden. Es ist aber auch ein `Runnable`, weil es `println()` auch ohne Parameterliste gibt:

```
Consumer<String> out = System.out::println;
out.accept( "Kates kurze Kleider" );
Runnable out = System.out::println;
out.run();
```

Ist eine Hauptmethode mit `main(String... args)` deklariert, so ist das auch ein `Runnable`:

```
Runnable r = JavaApplication1::main;
```

Anders wäre das bei `main(String[])`, hier ist ein Parameter zwingend, doch ein Vararg kann auch leer sein.

Anstatt den Namen einer Referenzvariablen zu wählen, kann auch `this` das Objekt beschreiben, und auch `super` ist möglich. `this` ist praktisch, wenn die Implementierung einer funktionalen Schnittstelle auf eine Methode der eigenen Klasse delegieren möchte. Wenn zum Beispiel eine lokale Methode `compareTrimmed(…)` in der Klasse existieren würde, in der auch der Lambda-Ausdruck steht, und wenn diese Methode als `Comparator` in `Arrays.sort(…)` verwendet werden sollte, könnte es heißen: `Arrays.sort(words, this::compareTrimmed)`.

Hinweis

Es ist nicht möglich eine spezielle Methode über die Methodenreferenz auszuwählen. Eine Angabe wie `String::valueOf` oder `Arrays::sort` ist relativ breit – bei Letzterem wählt der Compiler eine der 18 passenden überladenen Methoden aus. Da kann es passieren, dass der Compiler eine falsche Methode auswählt. In dem Fall muss ein expliziter Lambda-Ausdruck eine Mehrdeutigkeit auflösen. Bei generischen Typen kann zum Beispiel `List<String>::length` oder auch `List::length` stehen, auch hier erkennt der Compiler wieder alles selbst.

Was soll das alles?

Einem Einsteiger in die Sprache Java wird dieses Sprache-Feature wie der größte Zauber auf Erden vorkommen, und auch Java-Profis bekommen hier zittrige Finger, entweder vor Furcht oder Aufregung … In der Vergangenheit musste in Java sehr viel Code explizit geschrieben werden, aber mit diesen neuen Methoden-Referenzen erkennt und macht der Compiler vieles von selbst.

Nützlich wird diese Eigenschaft mit den funktionalen Bibliotheken bei der Stream-API aus Java 8, die in Abschnitt 4.11 vorgestellt wird. Hier nur ein kurzer Vorgeschmack:


```
Object[] words = { " ", '3', null, "2", 1, "" };
Arrays.stream( words )
      .filter( Objects::nonNull )
      .map( Objects::toString )
      .map( String::trim )
      .filter( s -> ! s.isEmpty() )
      .map( Integer::parseInt )
      .sorted()
      .forEach( System.out::println );   // 1 2 3
```

1.2.4 Konstruktor-Referenz

Um ein Objekt aufzubauen, nutzen wir den `new`-Operator. Wenn wir `new` nutzen, dann wird ein Konstruktor aufgerufen, und optional lassen sich Argumente an den Konstruktor übergeben. Die Java-API deklariert aber auch Typen, von denen sich keine Exemplare mit `new` aufbauen lassen. Stattdessen gibt es Erzeuger, deren Aufgabe es ist, Objekte aufzubauen. Die Erzeuger können statische oder auch nichtstatische Methoden sein:

Konstruktor …	… erzeugt	Erzeuger …	… baut
`new Integer( "1" )`	`Integer`	`Integer.valueOf( "1" )`	`Integer`
`new File( "dir" )`	`File`	`Paths.get( "dir" )`	`Path`
`new BigInteger( val )`	`BigInteger`	`BigInteger.valueOf( val )`	`BigInteger`

Tabelle 1.5 Beispiele für Konstruktoren und Erzeuger-Methoden

Beide, Konstruktoren und Erzeuger, lassen sich als spezielle Funktionen sehen, die von einem Typ in einen anderen Typ konvertieren. Damit eignen sie sich perfekt für Transformationen, und in einem Beispiel haben wir das schon eingesetzt:

```
Arrays.stream( words )
      . …
      .map( Integer::parseInt )
      . …
```

`Integer.parseInt(string)` ist eine Methode, die sich einfach mit einer Methoden-Referenz fassen lässt, und zwar als `Integer::parseInt`. Aber was ist mit Konstruktoren? Auch sie transformieren! Statt `Integer.parseInt(string)` hätte ja auch `new Integer(string)` eingesetzt werden können.

Wo Methoden-Referenzen statische Methoden und Objektmethoden angeben können, bieten *Konstruktor-Referenzen* die Möglichkeit, Konstruktoren anzugeben, sodass diese als Erzeuger an anderer Stelle übergeben werden können. Damit lassen sich elegant Konstruktoren als Erzeuger angeben, und zwar auch von einer Klasse, die nicht über Erzeugermethoden verfügt. Wie auch

bei Methoden-Referenzen spielt eine funktionale Schnittstelle eine entscheidende Rolle, doch dieses Mal ist es die Methode der funktionalen Schnittstelle, die mit ihrem Aufruf zum Konstruktor-Aufruf führt. Wo syntaktisch bei Methoden-Referenzen rechts vom Doppelpunkt ein Methodenname steht, ist dies bei Konstruktor-Referenzen ein `new`.[11] Also ergibt sich alternativ zu

```
.map( Integer::parseInt )       // Methode Integer.parseInt(String)
```

in unserem Beispiel das Ergebnis mittels:

```
.map( Integer::new )            // Konstruktor Integer(String)
```

Mit der Konstruktor-Referenz gibt es also vier Möglichkeiten, funktionale Schnittstellen zu implementieren; die drei verbleibenden Varianten sind Lambda-Ausdrücke, Methoden-Referenzen und klassische Implementierung über eine Klasse.

Beispiel

Die funktionale Schnittstelle sei:

```
interface DateFactory { Date create(); }
```

Die folgende Konstruktor-Referenz bindet den Konstruktor an die Methode `create()` der funktionalen Schnittstelle:

```
DateFactory factory = Date::new;
System.out.print( factory.create() ); // zum Beispiel Sat Dec 29 09:56:35 CET 2012
```

Beziehungsweise die letzten beiden Zeilen zusammengefasst:

```
System.out.println( ((DateFactory)(Date::new)).create() );
```

Soll nur der Standard-Konstruktor aufgerufen werden, muss die funktionale Schnittstelle nur eine Methode besitzen, die keinen Parameter besitzt und etwas zurückliefert. Der Rückgabetyp der Methode muss natürlich mit dem Klassentyp zusammenpassen. Das gilt für den Typ `DateFactory` aus unserem Beispiel. Doch es geht noch etwas generischer, zum Beispiel mit der vorhandenen funktionalen Schnittstelle `Supplier`, wie wir gleich sehen werden.

In der API finden sich oftmals Parameter vom Typ `Class`, die als Typangabe dazu verwendet werden, dass die Methode mit `newInstance()` Exemplare bilden kann. Der Einsatz von `Class` lässt sich durch eine funktionale Schnittstelle ersetzen, und Konstruktor-Referenzen lassen sich an Stelle von `Class`-Objekten übergeben.

Standard- und parametrisierte Konstruktoren

Beim Standard-Konstruktor hat die Methode nur eine Rückgabe, bei einem parametrisierten Konstruktor muss die Methode der funktionalen Schnittstelle natürlich über eine kompatible Parameterliste verfügen:

11 Da `new` ein Schlüsselwort ist, kann keine Methode so heißen; der Identifizierer ist also sicher.

Konstruktor	Date()	Date(long t)
Kompatible funktionale Schnittstelle	`interface DateFactory { Date create(); }`	`interface DateFactory { Date create(long t); }`
Konstruktor-Referenz	`DateFactory factory = Date::new;`	`DateFactory factory = Date::new;`
Aufruf	`factory.create();`	`factory.create(1);`

Tabelle 1.6 Standard- und parametrisierter Konstruktor mit korrespondierenden funktionalen Schnittstellen

Hinweis

Kommt die Typ-Inferenz des Compilers an ihre Grenzen, sind zusätzliche Typinformationen gefordert. In diesem Fall werden hinter dem Doppelpunkt in eckigen Klammen weitere Angaben gemacht, etwa `Klasse::<Typ1, Typ2>new`.

Nützliche vordefinierte Schnittstellen für Konstruktor-Referenzen

Die für einen Standard-Konstruktor passende funktionale Schnittstelle muss eine Rückgabe besitzen und keinen Parameter annehmen; die funktionale Schnittstelle für einen parametrisierten Konstruktor muss eine entsprechende Parameterliste haben. Es kommt nun häufig vor, dass der Konstruktor ein Standard-Konstruktor ist oder genau einen Parameter annimmt. Hier ist es vorteilhaft, dass für diese beiden Fälle die Java-API zwei praktische (generisch deklarierte) funktionale Schnittstellen mitbringt:

Funktionale Schnittstelle	Funktions-Deskriptor	Abbildung	Passt auf
`Supplier<T>`	`T get()`	() -> T	Standard-Konstruktor
`Function<T, R>`	`R apply(T t)`	(T) -> R	einfacher parametrisierter Konstruktor

Tabelle 1.7 Vorhandene funktionale Schnittstellen als Erzeuger

Beispiel

Die funktionale Schnittstelle `Supplier<T>` hat eine `T get()`-Methode, die wir mit dem Standard-Konstruktor von `Date` verbinden können:

```
Supplier<Date> factory = Date::new;
System.out.print( factory.get() );
```

Wir nutzen `Supplier` mit dem Typparameter `Date`, was den parametrisierten Typ `Supplier<Date>` ergibt, und `get()` liefert folglich den Typ `Date`. Der Aufruf `factory.get()` führt zum Aufruf des Konstruktors.

Ausblick *

Besonders interessant werden die Konstruktor-Referenzen mit den neuen Bibliotheksmethoden von Java 8. Nehmen wir eine Liste vom Typ Zeitstempel an. Der Konstruktor `Date(long)` nimmt einen solchen Zeitstempel entgegen, und mit einem `Date`-Objekt können wir Vergleiche vornehmen, etwa ob ein Datum hinter einem anderen Datum liegt. Folgendes Beispiel listet alle Datumswerte auf, die nach dem 1.1.2012 liegen:

```
Long[] timestamps = { 2432558632L, 1455872986345L };
Date thisYear = new GregorianCalendar( 2012, Calendar.JANUARY, 1 ).getTime();
Arrays.stream( timestamps )
      .map( Date::new )
      .filter( thisYear::before )
      .forEach( System.out::println );  // Fri Feb 19 10:09:46 CET 2016
```

Die Konstruktor-Referenz `Date::new` hilft dabei, das `long` mit dem Zeitstempel in ein `Date`-Objekt zu konvertieren.

Denksportaufgabe

Ein Konstruktor kann als `Supplier` bzw. `Function` gelten. Problematisch sind mal wieder geprüfte Ausnahmen. Der Leser soll überlegen, ob der Konstruktor `URI(String str) throws URISyntaxException` über `URI::new` angesprochen werden kann.

1.2.5 Implementierung von Lambda-Ausdrücken

Als die Compilerentwickler einen Prototyp für Lambda-Ausdrücke bauten, setzten sie diese technisch mit inneren Klassen um. Doch das war nur in der Testphase, denn innere Klassen sind für die JVM komplette Klassen und relativ schwergewichtig. Das Laden und Initialisieren ist relativ teuer und würde bei den vielen kleinen Lambda-Ausdrücken einen großen Overhead darstellen. Daher nutzt die Implementierung in Java 8 ein Konstrukt aus Java 7, genannt *invokedynamic*. Das hat den großen Vorteil, dass die Laufzeitumgebung viel Gestaltungsraum in der Optimierung hat. Innere Klassen sind nur eine mögliche technische Umsetzung für Lambda-Ausdrücke, invokedynamic ist sozusagen die deklarative Variante, und innere Klassen sind die imperative. Letztendlich ist der Overhead mit invokedynamic gering, und Programmcode von inneren Klassen hin zu Lambda-Ausdrücken zu refaktorisieren führt zu kleinen Bytecodedateien. Von der Performance her unterscheiden sich Lambda-Ausdrücke und die Implementierung funktionaler Schnittstellen und Klassen nicht, eher ist die Optimierung auf der Seite der JVM zu finden, die es mit weniger Klassendateien zu tun hat. Umgekehrt bedeutet das auch, wenn Entwickler ihre alte vorhandene Implementierung von funktionalen Schnittstellen durch Lambda-Ausdrücke ersetzen, wird der Bytecode kompakter, da ein kleines invokedynamic viel kürzer ist als komplexe neue Klassendateien.

1.2.6 Funktionale Programmierung mit Java

Programmierparadigmen: imperativ oder deklarativ

In irgendeiner Weise muss ein Entwickler sein Problem in Programmform beschreiben, damit der Computer es letztendlich ausführen kann. Hier gibt es verschiedene Beschreibungsformen, die wir *Programmierparadigmen* nennen. Bisher haben wir uns immer mit der imperativen Programmierung beschäftigt, bei der Anweisungen im Mittelpunkt stehen. Wir haben im Deutschen den Imperativ, also die Befehlsform, die sehr gut mit dem Programmierstil vergleichbar ist, denn es handelt sich in beiden Fällen um Anweisungen der Art »tue dies, tue das«. Diese »Befehle« mit Variablen, Fallunterscheidungen, Sprüngen beschreiben das Programm und den Lösungsweg.

Zwar ist imperative Programmierung die technisch älteste, aber nicht die einzige Form, Programme zu beschreiben; es gibt daneben die deklarative Programmierung, die nicht das Wie zur Problemlösung beschreibt, sondern das Was, also was eigentlich gefordert ist, ohne sich in genauen Abläufen zu verstricken. Auf den ersten Blick klingt das abstrakt, aber für jeden, der schon einmal

- einen Selektion wie **.html* auf der Kommandozeile/im Explorer-Suchfeld getätigt,
- eine Datenbankabfrage mit SQL geschrieben,
- eine XML-Selektion mit XQuery genutzt,
- ein Build-Skript mit Ant oder make formuliert oder
- eine XML-Transformation mit XSLT beschrieben hat,

wird das Prinzip kennen.

Bleiben wir kurz bei SQL, um einen Punkt deutlich zu machen. Natürlich ist im Endeffekt die Abarbeitung der Tabellen und Auswertungen der Ergebnisse von der CPU rein imperativ, doch es geht um die Programmbeschreibung auf einem höheren Abstraktionsniveau. Deklarative Programme sind üblicherweise wesentlich kürzer, und damit kommen weitere Vorteile wie leichtere Erweiterbarkeit, Verständlichkeit ins Spiel. Da deklarative Programme oftmals einen mathematischen Hintergrund haben, lassen sich die Beschreibungen leichter formal in ihrer Korrektheit beweisen.

Deklarative Programmierung ist ein Programmierstil, und eine deklarative Beschreibung braucht eine Art »Ablaufumgebung«, denn SQL kann zum Beispiel keine CPU direkt ausführen. Aber anstatt nur spezielle Anwendungsfälle wie Datenbank- oder XML-Abfragen zu behandeln, können auch typische Algorithmen deklarativ formuliert werden, und zwar mit funktionaler Programmierung. Damit sind imperative Programme und funktionale Programme gleich mächtig in ihren Möglichkeiten.

Funktionale Programmierung und funktionale Programmiersprachen

Bei der funktionalen Programmierung stehen Funktionen im Mittelpunkt und ein im Idealfall zustandsloses Verhalten, in dem viel mit Rekursion gearbeitet wird. Ein typisches Beispiel ist die Berechnung der Fakultät. Es ist n! = 1 × 2 × 3 × ... × n, und mit Schleifen und Variablen, dem imperativen Weg, sieht sie so aus:

```
public static int factorial( int n ) {
  int result = 1;
  for ( int i = 1; i <= n; i++ )
    result *= i;
  return result;
}
```

Deutlich sind die vielen Zuweisungen und die Fallunterscheidung durch die Schleife abzulesen, die typischen Indikatoren für imperative Programme. Der Schleifenzähler erhöht sich, damit kommt Zustand in das Programm, denn der aktuelle Index muss ja irgendwo im Speicher gehalten werden. Bei der rekursiven Variante ist das ganz anders, hier gibt es keine Zuweisungen im Programm, und die Schreibweise erinnert an die mathematische Definition:

```
public static int factorial( int n ) {
  return n == 0 ? 1 : n * factorial( n - 1 );
}
```

Mit der funktionalen Programmierung haben wir eine echte Alternative zur imperativen Programmierung. Die Frage ist nur: Mit welcher Programmiersprache lassen sich funktionale Programme schreiben? Im Grunde mit jeder höheren Programmiersprache! Denn funktional zu programmieren ist ja ein Programmierstil, und Java unterstützt funktionale Programmierung, wie wir am Beispiel mit der Fakultät ablesen können. Da das im Prinzip schon alles ist, stellt sich die Frage, warum funktionale Programmierung einen so schweren Stand hat und bei den Entwicklern gefürchtet ist. Das hat mehrere Gründe:

- **Lesbarkeit**: Am Anfang der funktionalen Programmiersprachen steht historisch LISP aus dem Jahr 1958, eine sehr flexible, aber ungewohnt zu lesende Programmiersprache. Unsere Fakultät sieht in LISP so aus:

  ```
  (defun factorial (n) (if (= n 1) 1 (* n (factorial (- n 1)))))
  ```

  Die ganzen Klammern machen die Programme nicht einfach lesbar, und die Ausdrücke stehen in der Präfix-Notation `- n 1` statt der üblichen Infix-Notation `n - 1`. Bei anderen funktionalen Programmiersprachen ist es anders, dennoch führt das zu einem gewissen Vorurteil, dass alle funktionalen Programmiersprachen schlecht lesbar sind.
- **Performance und Speicherverbrauch**: Ohne clevere Optimierungen von Seiten des Compilers und der Laufzeitumgebung führen insbesondere rekursive Aufrufe zu prall gefüllten Stacks und schlechter Laufzeit.

- **Rein funktional**: Es gibt funktionale Programmiersprachen, die als »rein« oder »pur« bezeichnet werden und keine Zustandsänderungen erlauben. Die Entwicklung von Ein-/Ausgabeoperationen oder simplen Zufallszahlen ist ein großer Akt, der für normale Entwickler nicht mehr nachvollziehbar ist. Die Konzepte sind kompliziert, doch zum Glück sind die meisten funktionalen Sprachen nicht so rein und erlauben Zustandsänderungen, nur Programmierer versuchen genau diese Zustandsänderungen zu vermeiden, um sich nicht die Nachteile damit einzuhandeln.
- **Funktional mit Java**: Wenn es darum geht, nur mit Funktionen zu arbeiten, kommen Entwickler schnell an einen Punkt, an dem Funktionen andere Funktionen als Argumente übergeben oder Funktionen zurückgeben. So etwas lässt sich in Java in der traditionellen Syntax nur sehr umständlich schreiben. Dies führt dazu, dass alles so unlesbar wird, dass der ganze Vorteil der kompakten deklarativen Schreibweise verloren geht.

Aus heutiger Sicht stellt sich eine Kombination aus beiden Konzepten als zukunftsweisend dar. Mit der in Java 8 eingeführten Schreibweise der Lambda-Ausdrücke sind funktionale Programme kompakt und relativ gut lesbar, und die JVM hat gute Optimierungsmöglichkeiten. Java ermöglicht beide Programmierparadigmen, und Entwickler können den Weg wählen, der für eine Problemlösung gerade am besten ist. Diese Mehrdeutigkeit schafft natürlich auch Probleme, denn immer wenn es mehrere Lösungswege gibt, entstehen Auseinandersetzungen um die beste der Varianten – und hier kann von Entwickler zu Entwickler eine konträre Meinung herrschen. Funktionale Programmierung hat unbestrittene Vorteile, und das wollen wir uns genau anschauen.

Funktionale Programmierung in Java am Beispiel vom Comparator

Funktionale Programmierung hat auch daher etwas Akademisches, weil in den Köpfen der Entwickler oftmals dieses Programmierparadigma nur mit mathematischen Funktionen in Verbindung gebracht wird. Und die wenigsten werden tatsächlich Fakultät oder Fibonacci-Zahlen in Programmen benötigen und daher schnell funktionale Programmierung beiseitelegen. Doch diese Vorurteile sind unbegründet, und es ist hilfreich, funktionale Programmierung gedanklich von der Mathematik zu lösen, denn die allermeisten Programme haben nichts mit mathematischen Funktionen im eigentlichen Sinne zu tun, wohl aber viel stärker mit formal beschriebenen Methoden.

Betrachten wir erneut unser Beispiel aus der Einleitung, die Sortierung von Strings, diesmal aus der Sicht eines funktionalen Programmierers. Ein `Comparator` ist eine einfache »Funktion«, mit zwei Parametern und einer Rückgabe. Diese »Funktion« (realisiert als Methode) wiederum wird an die `sort(…)`-Methode übergeben. Alles das ist funktionale Programmierung, denn wir programmieren Funktionen und übergeben sie. Drei Beispiele (Generics ausgelassen):

Code	Bedeutung
`Comparator c = (c1, c2) -> …`	Implementiert eine Funktion über einen Lambda-Ausdruck.
`Arrays.sort(T[] a, Comparator c)`	Nimmt eine Funktion als Argument an.
`Collections.reverseOrder(Comparator cmp)`	Nimmt eine Funktion an und liefert auch eine zurück.

Tabelle 1.8 Beispiele für Funktionen in der Übergabe und als Rückgabe

Funktionen selbst können in Java nicht übergeben werden, also helfen sich Java-Entwickler mit der Möglichkeit, die Funktionalität in eine Methode zu setzen, sodass die Funktion zum Objekt mit einer Methode wird, was die Logik realisiert. Lambda-Ausdrücke bzw. Methoden/Konstruktor-Referenzen geben eine kompakte Syntax ohne den Ballast, extra eine Klasse mit einer Methode schreiben zu müssen.

Der Typ `Comparator` ist eine funktionale Schnittstelle und steht für eine besondere Funktion mit zwei Parametern gleichen Typs und einer Ganzzahl-Rückgabe. Es gibt weitere funktionale Schnittstellen, die etwas flexibler sind als `Comparator`, in der Weise, dass etwa die Rückgabe statt `int` auch `double`oder etwas anderes sein kann.

Lambda-Ausdrücke als Funktionen sehen

Wir haben gesehen, dass sich Lambda-Ausdrücke in einer Syntax formulieren lassen, die folgende allgemeine Form hat:

```
'(' LambdaParameter ')' '->' '{' Anweisungen '}'
```

Der Pfeil macht gut deutlich, dass wir es bei Lambda-Ausdrücken mit Funktionen zu tun haben, die etwas abbilden. Im Fall vom `Comparator` ist es eine Abbildung von zwei Strings auf eine Ganzzahl – in eine etwas mathematischere Notation gepackt: (String, String) → int.

Beispiel

Methoden gibt es mit und ohne Rückgabe und mit und ohne Parameter. Genauso ist das mit Lambda-Ausdrücken. Ein paar Beispiele in Java-Code mit ihren Abbildungen sehen Sie in Tabelle 1.9:

Lambda-Ausdruck	Abbildung
`(int a, int b) -> a + b`	(int, int) → int
`(int a) -> Math.abs( a )`	(int) → int
`(String s) -> s.isEmpty()`	(String) → boolean

Tabelle 1.9 Lambda-Ausdrücke und was sie als Funktionen abbilden

Lambda-Ausdruck	Abbildung
`(Collection c) -> c.size()`	(Collection) → int
`() -> Math.random()`	() → double
`(String s) -> { System.out.print( s ); }`	(String) → void
`() -> {}`	() → void

Tabelle 1.9 Lambda-Ausdrücke und was sie als Funktionen abbilden (Forts.)

Begriff: Funktion versus Methode

Die Java-Sprachdefinition kennt den Begriff »Funktion« nicht, sondern spricht nur von Methoden. Methoden hängen immer an Klassen, und das heißt, dass Methoden immer an einem Kontext hängen. Das ist zentral bei der Objektorientierung, da Methoden auf Attribute lesend und schreibend zugreifen können. Lambda-Ausdrücke wiederum realisieren Funktionen, die erst einmal ihre Arbeitswerte rein aus den Parametern beziehen, sie hängen nicht an Klassen und Objekten. Der Gedanke bei funktionalen Programmiersprachen ist der, ohne Zustände auszukommen, also Funktionen so clever anzuwenden, dass sie ein Ergebnis liefern. Funktionen geben für eine spezifische Parameterkombination immer dasselbe Ergebnis zurück, unabhängig vom Zustand des umgebenden Gesamtprogramms.

1.2.7 Funktionale Schnittstelle aus dem java.util.function-Paket

Funktionen realisieren Abbildungen, und da es verschiedene Arten von Abbildungen geben kann, bietet die Java-Standardbibliothek im Paket `java.util.function` für die häufigsten Fälle funktionale Schnittstellen an. Ein erster Überblick:

Schnittstelle	Abbildung
`Consumer<T>`	(T) → void
`DoubleConsumer`	(double) → void
`BiConsumer<T, U>`	(T, U) → void
`Supplier<T>`	() → T
`BooleanSupplier`	() → boolean
`Predicate<T>`	(T) → boolean
`LongPredicate`	(long) → boolean

Tabelle 1.10 Beispiele einiger vordefinierter funktionaler Schnittstellen

Schnittstelle	Abbildung
`BiPredicate<T, U>`	(T, U) → boolean
`Function<T, R>`	(T) → R
`LongToDoubleFunction`	(long) → double
`BiFunction<T, U, R>`	(T, U) → R
`UnaryOperator<T>`	(T) → T
`DoubleBinaryOperator`	(double) → boolean

Tabelle 1.10 Beispiele einiger vordefinierter funktionaler Schnittstellen (Forts.)

Blöcke mit Code und die funktionale Schnittstelle java.util.function.Consumer

Anweisungen von Code lassen sich in eine Methode eines Objekts setzen und auf diese Weise weitergeben. Das ist eine häufige Notwendigkeit, für die das Paket `java.util.function` eine einfache funktionale Schnittstelle `Consumer` vorgibt, die einen Konsumenten repräsentiert, der Daten annimmt und dann verbraucht (konsumiert).

```
interface java.util.function.Consumer<T>
```

- `void accept(T t)`
  Führt Operationen mit der Übergabe `t` durch.
- `default Consumer<T> andThen(Consumer<? super T> after)`
  Liefert einen neuen `Consumer`, der erst den aktuellen `Consumer` ausführt und danach `after`.

Die `accept(…)`-Methode bekommt ein Argument – wobei die Implementierung natürlich nicht zwingend darauf zurückgreifen muss – und liefert keine Rückgabe. Transformationen sind damit nicht möglich, denn nur über Umwege kann der Konsument die Ergebnisse speichern, und dafür ist die Schnittstelle nicht gedacht. `Consumer`-Typen sind eher gedacht als Endglied einer Kette, in der zum Beispiel Dateien in eine Datei geschrieben werden, die vorher verarbeitet wurden. Diese Seiteneffekte sind beabsichtigt, da sie nach einer Kette von seiteneffektfreien Operationen stehen.

Immer repräsentieren Konsumenten Code, und eine API kann nun einfach einen Codeblock nach der Art `doSomethingWith(myConsumer)` annehmen, um ihn etwa in einem Hintergrund-Thread abzuarbeiten oder wiederholend auszuführen, oder kann ihn nach einer erlaubten Maximaldauer abbrechen oder die Zeit messen oder, oder, oder …

Beispiel

Implementiere einen `Consumer`-Wrapper, der die Ausführungszeit eines anderen Konsumenten loggt:


```
import java.util.function.*;
import java.util.logging.Logger;
class Consumers {
  public static <T> Consumer<T> measuringConsumer( Consumer<T> block ) {
    return t -> {
      long start = System.nanoTime();
      block.accept( t );
      long duration = System.nanoTime() - start;
      Logger.getAnonymousLogger().info( "Ausführungszeit (ns): " + duration );
    };
  }
}
```

Folgender Aufruf zeigt die Nutzung:

```
Consumer<Void> wrap = measuringConsumer( Void -> System.out.println( "Test" ) );
wrap.accept( null );
```

Was wir hier implementiert haben, ist ein Beispiel vom Execute-around-Method-Muster, bei dem wir um einen Block Code noch etwas anderes legen.

Typ Consumer in der API

In der Java-API zeigt sich der Typ `Consumer` in der Regel als Argument einer Methode `forEach(Consumer)`, die Datenquellen abläuft und für jedes Element `accept(…)` aufruft. Interessant ist die Methode am Typ `Iterable`, denn die wichtigen `Collection`-Datenstrukturen wie `ArrayList` implementieren diese Schnittstelle. So lässt sich einfach über alle Daten laufen und ein Stück Code für jedes Element ausführen. Auch `Iterator` hat eine vergleichbare Methode, da heißt sie `forEachRemaining(Consumer)` – das »Remaining« macht deutlich, dass der `Iterator` schon ein paar `next()`-Aufrufe erlebt haben könnte und die Konsumenten daher nicht zwingend die ersten Elemente mitbekommen.

Beispiel

Gib jedes Element einer Liste auf der Konsole aus:

```
Arrays.asList( 1, 2, 3, 4 ).forEach( System.out::println );
```

Gegenüber einem normalen Durchiterieren ist die funktionale Variante ein wenig kürzer im Code, aber sonst gibt es keinen Unterschied. Auch `forEach(…)` macht auf dem `Iterable` nichts anderes, als alle Elemente über den `Iterator` zu holen.

Supplier

Ein *Supplier* (auch *Provider* genannt) ist eine Fabrik und sorgt für Objekte. In Java deklariert das Paket `java.util.function` die funktionale Schnittstelle `Supplier` für Objektgeber:

```
interface java.util.function.Supplier<T>
```

- `T get()`
  Führt Operationen mit der Übergabe `t` durch.

Weitere statische oder Default-Methoden deklariert `Supplier` nicht. Was `get()` nun genau liefert, ist Aufgabe der Implementierung und ein Interna. Es können neue Objekte sein, immer die gleichen Objekte (Singleton) oder Objekte aus einem Cache.

Prädikate und java.util.function.Predicate

Ein Prädikat ist eine Aussage über einen Gegenstand, die wahr oder falsch ist. Die Frage mit `Character.isDigit('a')`, ob das Zeichen »a« eine Ziffer ist, wird mit falsch beantwortet – `isDigit` ist also ein Prädikat, weil es über einen Gegenstand, ein Zeichen, eine Wahrheitsaussage fällen kann.

Flexibler sind Prädikate, wenn sie als Objekte repräsentiert werden, weil sie dann an unterschiedliche Stellen weitergegeben werden können – etwa wenn über ein Prädikat bestimmt wird, das aus einer Sammlung gelöscht werden soll, oder wenn mindestens ein Element in einer Sammlung ist, welches ein Prädikat erfüllt.

Das `java.util.function`-Paket[12] deklariert eine flexible funktionale Schnittstelle `Predicate` auf folgende Weise:

```
interface java.util.function.Predicate<T>
```

- `boolean test(T t)`
  Führt einen Test auf `t` durch und liefert `true`, wenn das Kriterium erfüllt ist, sonst `false`.

Beispiel

Der Test, ob ein Zeichen eine Ziffer ist, kann durch Prädikat-Objekte nun auch anders durchgeführt werden:

```
Predicate<Character> isDigit = c -> Character.isDigit( c );
// kurz: Character::isDigit
System.out.println( isDigit.test('a') );  // false
```

Hätte es die Schnittstelle `Predicate` schon früher in Java 1.0 gegeben, hätte es der Methode `Character.isDigit(…)` gar nicht bedurft, es hätte auch ein `Predicate<Character>` als statische Variable in der Klasse `Character` geben können, sodass ein Test dann geschrieben würde als `Character.IS_DIGIT.test(…)` oder als Rückgabe von einer Methode `Predicate<Character> isDigit()` mit der Nutzung `Character.isDigit().test(…)`. Es ist daher gut möglich, dass sich in Zukunft die API

12 Achtung, in `javax.sql.rowset` gibt es ebenfalls eine Schnittstelle `Predicate`.

dahingehend verändert, dass Aussagen auf Gegenständen mit Wahrheitsrückgabe nicht mehr als Methoden bei den Klassen realisiert werden, sondern als Prädikat-Objekte angeboten werden. Aber Methoden-Referenzen geben zum Glück die Flexibilität, dass existierende Methoden problemlos als Lambda-Ausdrücke genutzt werden können, und so kommen wir wieder von Methoden zu Funktionen.

Typ Predicate in der API

Es gibt in der Java-API einige Stellen, an denen `Predicate`-Objekte genutzt werden:

- als Argument für Löschmethoden, um in Sammlungen Elemente zu spezifizieren, die gelöscht oder nach denen gefiltert werden soll
- bei den Default-Methoden der `Predicate`-Schnittstelle selbst, um Prädikate zu verknüpfen
- bei regulären Ausdrücken; ein `Pattern` liefert mit `asPredicte()` ein `Predicate` für Tests.
- in der Stream-API, bei der Objekte beim Durchlaufen des Stroms über ein Prädikat identifiziert werden, um sie etwa auszufiltern

Beispiel

Lösche aus einer Liste mit Zeichen alle, die Ziffern sind (es bleiben nur Zeichen übrig, etwa Buchstaben):

```
Predicate<Character> isDigit = Character::isDigit;
List<Character> list = new ArrayList<>( Arrays.asList( 'a', '1' ) );
list.removeIf( isDigit );
```

Auf diese Weise steht nicht die Schleife, sondern das Löschen im Vordergrund.

Default-Methoden von Predicate

Es gibt eine Reihe von Default-Methoden, die die funktionale Schnittstelle `Predicate` anbietet. Zusammenfassend:

```
interface java.util.function.Predicate<T>
```

- `default Predicate<T>negate()`
  Liefert vom aktuellen Prädikat eine Negation. Implementiert als `return t -> ! test(t);`.
- `default Predicate<T>and(Predicate<? super T> p)`
- `default Predicate<T>or(Predicate<? super T> p)`
  Und/Oder-verknüpfen das aktuelle Prädikat mit einem anderen Prädikat.
- `static <T> Predicate<T> isEqual(Object targetRef)`
  Liefert ein neues Prädikat, welches einen Gleichheitstest mit `targetRef` vornimmt, im Grunde `return ref -> Objects.equals(ref, targetRef)`.

Beispiel

Lösche aus einer Liste mit Zeichen alle die, die *keine* Ziffern sind:

```
Predicate<Character> isDigit = Character::isDigit;
Predicate<Character> isNotDigit = isDigit.negate();
List<Character> list = new ArrayList<>( Arrays.asList( 'a', '1' ) );
list.removeIf( isNotDigit );
```

Prädikate aus Pattern

Seit Java 8 liefert die `Pattern`-Methode `asPredicate()` ein `Predicate<String>`, sodass ein regulärer Ausdruck als Kriterium, zum Beispiel zum Filtern oder Löschen von Einträgen in Datenstrukturen, genutzt werden kann.

Beispiel

Lösche aus einer Liste alle Strings, die leer oder Zahlen sind:

```
List<String> list = new ArrayList<>( Arrays.asList( "Peaches", "", "25", "Geldof" ) );
list.removeIf( Pattern.compile( "\\d+" ).asPredicate().or( String::isEmpty ) );
System.out.println( list );           // [Peaches, Geldof]
```

Funktionen und die allgemeine funktionale Schnittstelle java.util.function.Function

Funktionen im Sinne der funktionalen Programmierung können in verschiedenen Bauarten vorkommen: mit Parameterliste/Rückgabe oder ohne. Doch im Grunde sind es Spezialformen, und die funktionale Schnittstelle `java.util.function.Function` ist die allgemeinste, die zu einem Argument ein Ergebnis liefert.

```
interface java.util.function.Function<T, R>
```

- `R apply(T t)`
  Wendet eine Funktion an, und liefert zur Eingabe `t` eine Rückgabe.

Beispiel

Eine Funktion zur Bestimmung des Absolutwerts:

```
Function<Double, Double> abs = a -> Math.abs( a );  // alternativ Math::abs
System.out.println( abs.apply( -12. ) );            // 12.0
```

Auch bei Funktionen ergibt sich für das API-Design ein Spannungsfeld, denn im Grunde müssen »Funktionen« nun gar nicht mehr als Methoden angeboten werden, sondern Klassen könnten sie auch als `Function`-Objekte anbieten. Doch da Methoden-Referenzen problemlos die Brücke von Methodennamen zu Objekten schlagen, fahren Entwickler mit klassischen Methoden ganz gut.

Typ Function in der API

Die Stream-API ist der größte Nutznießer vom `Function`-Typ. Es finden sich einige wenige Beispiele bei Objekt-Vergleichen (`Comparator`), im Paket für Nebenläufigkeiten und bei Assoziativspeichern. Im Abschnitt über die Stream-API werden wir daher viele weitere Beispiele kennenlernen.

Beispiel

Ein Assoziativspeicher soll als Cache realisiert werden, der zu Dateinamen den Inhalt assoziiert. Ist zu dem Schlüssel (dem Dateinamen) noch kein Inhalt vorhanden, soll dieser in den Assoziativspeicher gelegt werden.

```
class FileCache {
  private Map<String, byte[]> map = new HashMap<>();
  public byte[] getContent( String filename ) {
    return map.computeIfAbsent( filename, file -> {
      try {
        return Files.readAllBytes( Paths.get( file ) );
      } catch ( IOException e ) { throw new UncheckedIOException( e ); }
    } );
  }
}
```

Auf die Methode kommt Kapitel 15, »RESTful und SOAP-Web-Services«, noch einmal zurück, das Beispiel soll nur eine Idee geben, dass Funktionen an andere Funktionen übergeben werden – hier eine `Function<String, byte[]>` an `computeIfAbsent(…)`. Sobald an den Datei-Cache der Aufruf `getContent(String)`geht, wird dieser die `Map` fragen und wenn diese zu dem Schlüssel keinen Wert hat, wird sie den Lambda-Ausdruck auswerten, um zu dem Dateinamen den Inhalt zu liefern.

Getter-Methoden als Function über Methoden-Referenzen

Methoden-Referenzen gehören zu den stärksten Sprachmitteln von Java 8. In Kombination mit Gettern ist ein Muster abzulesen, welches oft in Code zu sehen ist. Zunächst noch einmal zur Wiederholung von `Function` und der Nutzung bei Methoden-Referenzen:

```
Function<String, String> func2a = (String s) -> s.toUpperCase();
Function<String, String> func2b = String::toUpperCase;

Function<Point, Double> func1a = (Point p) -> p.getX();
Function<Point, Double> func1b = Point::getX;

System.out.println( func2b.apply( "jocelyn" ) );         // JOCELYN
System.out.println( func1b.apply( new Point( 9, 0 ) ) ); // 9.0
```

Dass `Function` auf die gegebene Methoden-Referenz passt, ist auf den ersten Blick unverständlich, da die Signaturen von `toUpperCase()` und `getX()` keinen Parameter deklarieren, also im üblichen Sinne keine Funktionen sind, wo etwas reinkommt und wieder rauskommt. Wir haben es

hier aber mit einem speziellen Fall zu tun, denn die in der Methoden-Referenz genannten Methoden sind a) nicht statisch – wie `Math::max` – und b) ist auch keine Referenz – wie `System.out::print` – im Spiel, sondern hier wird der Compiler eine Objektmethode auf genau dem Objekt aufrufen, das als erstes Argument der funktionalen Schnittstelle übergeben wurde. (Diesen Satz bitte zweimal lesen.)

Damit ist `Function` ein praktischer Typ bei allen Szenarien, bei denen irgendwie über Getter Zustände erfragt werden, wie es etwa bei einem `Comparator` öfter vorkommt. Hier ist eine statische Methode – die Generics einmal ausgelassen – `Comparator<…> Comparator.comparing(Function<…> keyExtractor)` sehr nützlich.

Beispiel

Besorge eine Liste von Pakten, die vom Klassenlader zugänglich sind, und sortiere sie nach Namen:

```
List<Package> list = Arrays.asList( Package.getPackages() );
Collections.sort( list, Comparator.comparing( Package::getName ) );
System.out.println( list ); // [package java.io, … sun.util.locale …
```

Default-Methoden in Function

Die funktionale Schnittstelle schreibt nur eine Methode `apply(…)` vor, deklariert jedoch noch drei zusätzliche Default-Methoden:

```
interface java.util.function.Function<T,R>
```

- `static <T> Function<T,T> identity()`
  Liefert eine neue Funktion, die immer die Eingabe als Ergebnis liefert.
- `default <V> Function<T,V> andThen(Function<? super R,? extends V> after)`
  Entspricht `t -> after.apply(apply(t))`.
- `default <V> Function<V,R> compose(Function<? super V,? extends T> before)`
  Entspricht `v -> apply(before.apply(v))`.

Die Methoden `andThen(…)` und `compose(…)` unterscheiden sich also darin, in welcher Reihenfolge die Funktionen aufgerufen werden. Das Gute ist, dass die Parameternamen (»before«, »after«) klarmachen, was hier in welcher Reihenfolge aufgerufen wird, wenn auch »compose« selbst wenig aussagt.

Function versus Consumer/Predicate

Im Grunde lässt sich alles als `Function` darstellen, denn

- ein `Consumer<T>` lässt sich auch als `Function<T,Void>` verstehen (es geht etwas rein, aber nichts raus),
- ein `Predicate<T>` als `Function<T,Boolean>` und
- ein `Supplier<T>` als `Function<Void,T>`.

Dennoch erfüllen diese speziellen Typen ihren Zweck, denn je genauer der Typ, desto besser.

UnaryOperator

Es gibt auch eine weitere Schnittstelle im `java.util.function`-Paket, die `Function` spezialisiert, und zwar `UnaryOperator`. Ein `UnaryOperator` ist eine spezielle Funktion, bei der die Typen für »Eingang« und »Ausgang« gleich sind.

```
interface java.util.function.UnaryOperator<T>
extends Function<T,T>
```

- `static <T> UnaryOperator<T> identity()`
  Liefert den Identitäts-Operator, der alle Eingaben auf die Ausgaben abbildet.

Die generischen Typen machen deutlich, dass der Typ des Methodenparameters gleich dem Ergebnistyp ist. Bis auf `identity()` gibt es keine weitere Funktionalität, die Schnittstelle dient lediglich zur Typdeklaration.

An einigen Stellen der Java-Bibliothek kommt dieser Typ auch vor, etwa bei der Methode `replaceAll(UnaryOperator)` der `List`-Typen.

Beispiel

Verdopple jeden Eintrag in der Liste:

```
List<Integer> list = Arrays.asList( 1, 2, 3 );
list.replaceAll( e -> e * 2 );
System.out.println( list );  // [2, 4, 6]
```

Ein bisschen Bi …

Bi ist eine bekannte lateinische Vorsilbe für »zwei«, was übertragen auf die Typen aus `java.util.function` bedeutet, das statt eines Arguments zwei übergeben werden können.

Typ	Schnittstelle	Operation
Konsument	`Consumer<T>`	`void accept(T t)`
	`BiConsumer<T,U>`	`void accept(T t, U u)`
Funktion	`Function<T,R>`	`R apply(T t)`
	`BiFunction<T,U,R>`	`R apply(T t, U u)`
Prädikat	`Predicate<T>`	`boolean test(T t)`
	`BiPredicate<T,U>`	`boolean test(T t, U u)`

Tabelle 1.11 Ein-/Zwei-Argument-Methoden im Vergleich

Die Bi-Typen haben mit den Nicht-Bi-Typen keine Typbeziehung.[13]

BiConsumer

Der `BiConsumer` deklariert die Methode `accept(T, U)` mit zwei Parametern, die jeweils unterschiedliche Typen tragen können. Haupteinsatzpunkt des Typs in der Java-Standardbibliothek sind Assoziativspeicher, die Schlüssel und Werte an `accept(…)` übergeben. So deklariert `Map` die Methode:

```
interface java.util.Map<K,V>
```

- `default void forEach(BiConsumer<? super K,? super V> action)`
  Läuft den Assoziativspeicher ab und ruft auf jedem Schlüssel-Wert-Paar die `accept(…)`-Methode vom übergebenen `BiConsumer` auf.

Beispiel

Gib die Temperaturen der Städte aus:

```
Map<String, Integer> map = new HashMap<>();
map.put( "Manila", 25 );
map.put( "Leipzig", -5 );
map.forEach( (k,v) -> System.out.printf("%s hat %d Grad%n", k, v) );
```

Ein `BiConsumer` besitzt eine Default-Methode `andThen(…)`, wie auch der `Consumer` sie zur Verkettung deklariert.

```
interface java.util.function.BiConsumer<T,U>
```

- `default BiConsumer<T,U> andThen(BiConsumer<? super T,? super U> after)`
  Verknüpft den aktuellen `BiConsumer` mit `after` zu einem neuen `BiConsumer`.

BiFunction und BinaryOperator

Eine `BiFunction` ist eine Funktion mit zwei Argumenten, während eine normale `Function` nur ein Argument annimmt.

Beispiel

Nutzung von `Function` und `BiFunction` mit Methoden-Referenzen:

```
Function<Double, Double> sign = Math::abs;
BiFunction<Double, Double, Double> max = Math::max;
```

Die Java-Bibliothek greift viel öfter auf `Function` zurück als auf `BiFunction`. Der häufigste Einsatz findet sich in der Standardbibliothek rund um Assoziativspeicher, bei denen Schlüssel und Wert an eine `BiFunction` übergeben werden.

13 Irgendwie finden das manche Leser lustig …

Beispiel

Konvertiere alle assoziierten Werte einer `HashMap` nach Großschreibung:

```
Map<Integer, String> map = new HashMap<>();
map.put( 1, "eins" ); map.put( 2, "zwEi" );
System.out.println( map );  // {1=eins, 2=zwEi}
BiFunction<Integer, String, String> func = (k, v) -> v.toUpperCase();
map.replaceAll( func );
System.out.println( map );  // {1=EINS, 2=ZWEI}
```

Ist bei einer `Function` der Typ derselbe, bietet die Java-API dafür den spezielleren Typ `UnaryOperator`. Sind bei einer `BiFunction` alle drei Typen gleich, bietet sich hier `BinaryOperator` an – zum Vergleich:

- `interface UnaryOperator<T> extends Function<T,T>`
- `interface BinaryOperator<T> extends BiFunction<T,T,T>`

Beispiel

`BiFunction` und `BinaryOperator`:

```
BiFunction<Double, Double, Double> max1 = Math::max;
BinaryOperator<Double>             max2 = Math::max;
```

`BinaryOperator` spielt bei so genannten Reduktionen eine große Rolle, wenn zum Beispiel wie bei `max` aus zwei Werten einer wird, auch das beleuchtet der Abschnitt über die Stream-API später genauer.

Die Schnittstelle `BiFunction` deklariert genau eine Default-Methode:

```
interface java.util.function.BiFunction<T,U,R>
extends Function<T, T>
```

- `default <V> BiFunction<T,U,V> andThen(Function<? super R,? extends V> after)`

`BinaryOperator` dagegen wartet mit zwei statischen Methoden auf:

```
public interface java.util.function.BinaryOperator<T>
extends BiFunction<T,T,T>
```

- `static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)`
- `static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)`
  Liefert einen `BinaryOperator`, der das Maximum/Minimum bezüglich eines gegebenen `comparator` liefert.

BiPredicate

Ein `BiPredicate` testet zwei Argumente und verdichtet sie zu einem Wahrheitswert. Wie `Predicate` deklariert auch `BiPredicate` drei Default-Methoden `and(…)`, `or(…)` und `negate(…)`, wobei natürlich eine statische `isEqual(…)`-Methode wie bei `Predicate` in `BiPredicate` fehlt. Für `BiPredicate` gibt es in der Java-Standardbibliothek nur eine Verwendung bei einer Methode zum Finden von Dateien – der Gebrauch ist selten, zudem ja auch ein Prädikat immer einer Funktion mit `boolean`-Rückgabe ist, sodass es eigentlich für diese Schnittstelle keine zwingende Notwendigkeit gibt.

Beispiel

Bei `BiXXX` und zwei Argumenten hört im Übrigen die Spezialisierung auf, es gibt keine Typen `TriXXX`, `QuardXXX`, … Das ist in der Praxis auch nicht nötig, denn zum einen kann oftmals eine Reduktion stattfinden, so ist etwa `max(1, 2,3)` gleich `max(1, max(2, 3))`, und zum anderen kann auch der Parametertyp eine Sammlung sein, wie in `Function<List<Integer>, Integer> max`.

Funktionale Schnittstellen mit Primitiven

Die bisher vorgestellten funktionalen Schnittstellen sind durch die generischen Typparameter sehr flexibel, aber was fehlt, sind Signaturen mit Primitiven – Java hat das »Problem«, dass Generics nur mit Referenztypen funktionieren, nicht aber mit primitiven Typen. Aus diesem Grund gibt es von fast allen Schnittstellen aus dem `function`-Paket vier Versionen: eine generische für beliebige Referenzen sowie Versionen für den Typ `int`, `long` und `double`. Die API-Designer wollten gerne die Wrapper-Typen außen vor lassen und gewisse primitive Typen unterstützen, auch aus Performance-Gründen, um nicht immer ein Boxing durchführen zu müssen.

Tabelle 1.12 gibt einen Überblick über die funktionalen Schnittstellen, die alle keinerlei Vererbungsbeziehungen zu anderen Schnittstellen haben.

Funktionale Schnittstelle	Funktions-Deskriptor
XXXSupplier	
BooleanSupplier	boolean getAsBoolean()
IntSupplier	int getAsInt()
LongSupplier	long getAsLong()
DoubleSupplier	double getAsDouble()
XXXConsumer	
IntConsumer	void accept(int value)
LongConsumer	void accept(long value)
DoubleConsumer	void accept(double value)

Tabelle 1.12 Spezielle funktionale Schnittstellen für primitive Werte

Funktionale Schnittstelle	Funktions-Deskriptor
ObjIntConsumer<T>	void accept(T t, int value)
ObjLongConsumer<T>	void accept(T t, long value)
ObjDoubleConsumer<T>	void accept(T t, double value)
XXXPredicate	
IntPredicate	boolean test(int value)
LongPredicate	boolean test(long value)
DoublePredicate	boolean test(double value)
XXXFunction	
DoubleToIntFunction	int applyAsInt(double value)
IntToDoubleFunction	double applyAsDouble(int value)
LongToIntFunction	int applyAsInt(long value)
IntToLongFunction	long applyAsLong(int value)
DoubleToLongFunction	long applyAsLong(double value)
LongToDoubleFunction	double applyAsDouble(long value)
IntFunction<R>	R apply(int value)
LongFunction<R>	R apply(long value)
DoubleFunction<R>	R apply(double value)
ToIntFunction<T>	int applyAsInt(T t)
ToLongFunction<T>	long applyAsLong(T t)
ToDoubleFunction<T>	double applyAsDouble(T t)
ToIntBiFunction<T,U>	int applyAsInt(T t, U u)
ToLongBiFunction<T,U>	long applyAsLong(T t, U u)
ToDoubleBiFunction<T,U>	double applyAsDouble(T t, U u)
XXXOperator	
IntUnaryOperator	int applyAsInt(int operand)
LongUnaryOperator	long applyAsLong(long operand)

Tabelle 1.12 Spezielle funktionale Schnittstellen für primitive Werte (Forts.)

Funktionale Schnittstelle	Funktions-Deskriptor
`DoubleUnaryOperator`	`double applyAsDouble(double operand)`
`IntBinaryOperator`	`int applyAsInt(int left, int right)`
`LongBinaryOperator`	`long applyAsLong(long left, long right)`
`DoubleBinaryOperator`	`double applyAsDouble(double left, double right)`

Tabelle 1.12 Spezielle funktionale Schnittstellen für primitive Werte (Forts.)

Statische und Default-Methoden

Einige generisch deklarierte funktionale Schnittstellen-Typen besitzen Default-Methoden bzw. statische Methoden, und Ähnliches findet sich auch bei den primitiven funktionalen Schnittstellen wieder:

- Die `XXXConsumer`-Schnittstellen deklarieren `default XXXConsumer andThen(XXXConsumer after)`, aber nicht die `ObjXXXConsumer`-Typen, sie besitzen keine Default-Methode.
- Die `XXXPredicate`-Schnittstellen deklarieren:
  - `default XXXPredicate negate()`
  - `default XXXPredicate and(XXXPredicate other)`
  - `default IntPredicate or(XXXPredicate other)`
- Jeder `XXXUnaryOperator` besitzt:
  - `default XXXUnaryOperator andThen(IntUnaryOperator after)`
  - `default XXXUnaryOperator compose(XXXUnaryOperator before)`
  - `static IntUnaryOperator identity()`
- `BinaryOperator` hat zwei statische Methoden `maxBy(…)` und `minBy(…)`, die es nicht in der primitiven Version `XXXBinaryOperator` gibt, da kein `Comparator` bei primitiven Vergleichen nötig ist.
- Die `XXXSupplier`-Schnittstellen deklarieren keine statische- oder Default-Methoden, genauso wie es auch `Supplier` nicht tut.

1.2.8 Optional ist keine Nullnummer

Java hat eine besondere Referenz, die Entwicklern die Haare zu Berge stehen lässt und die Grund für lange Debug-Stunden ist: die `null`-Referenz. Eigentlich sagt `null` nur aus: »nicht initialisiert«. Doch was `null` so problematisch macht, ist die `NullPointerExcpetion`, die durch referenzierte `null`-Ausdrücke ausgelöst wird.

Beispiel

Entwickler haben vergessen, das Attribut `location` mit einem Objekt zu initialisieren, sodass `setLocation(…)` fehlschlagen wird:


```
class Place {
  private Point2D location;
  public void setLocation( double longitude, double latitude ) {
    location.setLocation( longitude, latitude );  // ☠ NullPointerException
  }
}
```

Einsatz von null

Fehler dieser Art sind durch Tests relativ leicht aufzuspüren. Aber hier liegt nicht das Problem. Das eigentliche Problem ist, dass Entwickler allzu gerne die typenlose `null`[14] als magischen Sonderwert sehen, sodass sie neben »nicht initialisiert« noch etwas anderes bedeutet:

- Erlaubt die API in Argumenten für Methoden/Konstruktoren `null`, heißt das meistens »nutze einen Default-Wert« oder »nichts gegeben, ignorieren«.
- In Rückgaben von Methoden steht `null` oftmals für »nichts gemacht« oder »keine Rückgabe«. Im Gegensatz dazu kodieren andere Methoden wiederum mit der Rückgabe `null`, dass eine Operation erfolgreich durchlaufen wurde, und würden sonst zum Beispiel Fehlerobjekte zurückgegeben.[15]

Beispiel 1

Die Javadoc Methode `getTask(out, fileManager, diagnosticListener, options, classes, compilationUnits)` in der Schnittstelle `JavaCompiler` ist so ein Beispiel:

- `out`: »a writer for additional output from the compiler; use system.err **if null«**
- `fileManager`: »a file manager; **if null** use the compiler's standard filemanager«
- `diagnosticListener`: »a diagnostic listener; **if null** use the compiler's default method for reporting diagnostics«
- `options`: »compiler options, **null** means no options«
- `classes`: »names of classes to be processed by annotation processing, **null** means no class names«
- `compilationUnits`: »the compilation units to compile, null means no compilation units«

Alle Argumente können `null` sein, `getTask(null, null, null, null, null, null)` ist also ein korrekter Aufruf. Schön ist die API nicht, und besser wäre sie wohl mit einem Builder-Pattern gelöst.

Beispiel 2

Der `BufferedReader` erlaubt das zeilenweise Einlesen aus Datenquellen, und `readLine()` liefert `null`, wenn es keine Zeile mehr zu lesen gibt.

14 `null instanceof Typ` ist immer `false`.

15 Zum Glück wird `null` selten als Fehler-Identifikator genutzt, die Zeiten sind vorbei. Hier sind Ausnahmen die bessere Wahl, denn Fehler sind Ausnahmen im Programm.

Beispiel 3

Viel Irritation gibt es mit der API vom Assoziativspeicher. Eine gewöhnliche `HashMap` kann als assoziierten Wert `null` bekommen, doch `get(key)` liefert auch dann `null`, wenn es keinen assoziierten Wert gibt. Das führt zu einer Mehrdeutigkeit, da die Rückgabe von `get(…)` nicht verrät, ob es eine Abbildung auf `null` gibt oder ob der Schlüssel nicht vorhanden ist.

```
Map<Integer, String> map = new HashMap<>();
map.put( 0, null );
System.out.println( map.containsKey( 0 ) );      // true
System.out.println( map.containsValue( null ) ); // true
System.out.println( map.get( 0 ) );              // null
System.out.println( map.get( 1 ) );              // null
```

Kann die Map `null`-Werte enthalten, muss es immer ein Paar der Art `if(map.containsKey(key))`, gefolgt von `map.get(key)` geben. Am besten verzichten Entwickler auf `null` in Datenstrukturen.

Da `null` so viele Einsatzfälle hat und das Lesen der API-Dokumentation gerne übersprungen wird, sollte es zu einigen `null`-Einsätzen Alternativen geben. Manches Mal ist das einfach, etwa wenn die Rückgabe Sammlungen sind. Dann gibt es mit einer leeren Sammlung eine gute Alternative zu `null`. Das ist ein Spezialfall des so genannten Null-Object-Patterns und wird in Kapitel 15, »RESTful und SOAP-Web-Services«, näher beschrieben.

Fehler, die aufgrund einer `NullPointerException` entstehen, ließen sich natürlich komplett vermeiden, wenn immer ordentlich auf `null`-Referenzen getestet würde. Aber gerade die `null`-Prüfungen werden von Entwicklern gerne vergessen, da ihnen nicht bewusst ist oder sie nicht erwarten, dass eine Rückgabe `null` sein kann. Gewünscht ist ein Programmkonstrukt, bei dem explizit wird, dass ein Wert nicht vorhanden sein kann, sodass nicht `null` diese Rolle übernehmen muss. Wenn im Code lesbar ist, dass ein Wert optional ist, also vorhanden sein kann oder nicht, reduziert das Fehler.

Geschichte

Tony Hoare gilt als »Erfinder« der `null`-Referenz. Heute bereut er es und nennt die Entscheidung »my billion-dollar mistake«.[16]

16 Er sagt dazu: *»It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.«*
Unter *http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare* gibt es ein Video mit ihm und Erklärungen.

Optional-Typ

Seit Java 8 bietet die Java-API eine Art Container, der ein Element enthalten kann oder nicht. Wenn der Container ein Element enthält, ist es nie `null`. Dieser Container kann befragt werden, ob er ein Element enthält oder nicht. Eine `null` als Kennung ist somit überflüssig.

Beispiel

```
Optional<String> opt1 = Optional.of( "Aitazaz Hassan Bangash" );
System.out.println( opt1.isPresent() );   // true
System.out.println( opt1.get() );         // Aitazaz Hassan Bangash
Optional<String> opt2 = Optional.empty();
System.out.println( opt2.isPresent() );   // false
// opt2.get() -> java.util.NoSuchElementException: No value present
Optional<String> opt3 = Optional.ofNullable( "Malala" );
System.out.println( opt3.isPresent() );   // true
System.out.println( opt3.get() );         // Malala
Optional<String> opt4 = Optional.ofNullable( null );
System.out.println( opt4.isPresent() );   // false
// opt4.get() -> java.util.NoSuchElementException: No value present
```

```
final class java.lang.Optional<T>
```

- `static <T> Optional<T> empty()`
  Liefert ein leeres `Optional`-Objekt.
- `boolean isPresent()`
  Liefert wahr, wenn dieses `Optional` einen Wert hat, sonst ist wie im Fall von `empty()` die Rückgabe `false`.
- `static <T> Optional<T> of(T value)`
  Baut ein neues `Optional` mit einem Wert auf, der nicht `null` sein darf; andernfalls gibt es eine `NullPointerException`. `null` in das `Optional` hineinzubekommen, geht also nicht.
- `static <T> Optional<T> ofNullable(T value)`
  Erzeugt ein `Optional` mit dem Wert, wenn dieser ungleich `null` ist, bei `null` ist die Rückgabe ein `empty()`-`Optional`.
- `T get()`
  Liefert den Wert. Handelt es sich um ein `Optional empty()`, folgt eine `NoSuchElementException`.
- `T orElse(T other)`
  Liefert den Wert vom `Optional` und, wenn dieser leer ist, `other`.

Weiterhin überschreibt `Optional` die Methoden `equals(…)`, `toString()` und `hashCode()` – 0, wenn kein Wert gegeben ist, sonst Hashcode vom Element – und ein paar weitere Methoden, die wir uns später anschauen.

Hinweis

Intern `null` zu verwenden hat zum Beispiel den Vorteil, dass die Objekte serialisiert werden können. `Optional` implementiert `Serializable` *nicht*, daher sind `Optional`-Attribute nicht serialisierbar, können also etwa nicht im Fall von Remote-Aufrufen mit RMI übertragen werden. Auch die Abbildung auf XML oder auf Datenbanken ist umständlicher, wenn nicht JavaBean-Properties herangezogen werden, sondern die internen Attribute.

Ehepartner oder nicht?

`Optional` wird also dazu verwendet, im Code explizit auszudrücken, ob ein Wert vorhanden ist oder nicht. Das gilt auf beiden Seiten: Der Erzeuger muss explizit `ofXXX(…)` aufrufen und der Nutzer explizit `isPresent()` oder `get()`. Beide Seiten sind sich also bewusst, dass sie es mit einem Wert zu tun haben, der optional ist, also existieren kann oder nicht. Wir wollen das in einem Beispiel nutzen und zwar für eine Person, die einen Ehepartner haben kann:

Listing 1.7 com/tutego/insel/java/lang/Person.java, Person

```
public class Person {
  private Person spouse;

  public void setSpouse( Person spouse ) {
    this.spouse = Objects.requireNonNull( spouse );
  }

  public void removeSpouse() {
    spouse = null;
  }

  public Optional<Person> getSpouse() {
    return Optional.ofNullable( spouse );
  }
}
```

In diesem Beispiel ist `null` für die interne Referenz auf den Partner möglich; diese Kodierung soll aber nicht nach außen gelangen. Daher liefert `getSpouse()` nicht direkt die Referenz, sondern es kommt `Optional` zum Einsatz und drückt aus, ob eine Person einen Ehepartner hat oder nicht. Auch bei `setSpouse(…)` akzeptieren wir kein `null`, denn `null`-Argumente sollten soweit wie möglich vermieden werden. Ein `Optional` ist hier nicht angemessen, weil es ein Fehler ist, `null` zu übergeben. Zusätzlich sollte natürlich die Javadoc an `setSpouse (…)` dokumentieren, dass ein `null`-Argument zu einer `NullPointerException` führt. Daher passt `Optional` als Parametertyp nicht.

Listing 1.8 com/tutego/insel/java/lang/OptionalDemo.java, main()

```
Person heinz = new Person();
System.out.println( heinz.getSpouse().isPresent() ); // false
Person eva = new Person();
heinz.setSpouse( eva );
System.out.println( heinz.getSpouse().isPresent() ); // true
System.out.println( heinz.getSpouse().get() ); // com/…/Person
heinz.removeSpouse();
System.out.println( heinz.getSpouse().isPresent() ); // false
```

Primitive optionale Typen

Während Referenzen `null` sein können, und auf diese Weise das Nichtvorhandensein anzeigen, ist das bei primitiven Datentypen nicht so einfach. Wenn eine Methode ein `boolean` zurückgibt, bleibt neben `true` und `false` nicht viel übrig, und ein »nicht zugewiesen« wird dann doch gerne wieder über einen `Boolean` verpackt und auf `null` getestet. Gerade bei Ganzzahlen gibt es immer wieder Rückgaben wie –1.[17] Das ist bei den folgenden Beispielen der Fall:

- Wenn bei `InputStream`s `read(…)` keine Eingaben mehr kommen, wird `-1` zurückgegeben.
- `indexOf(Object)` von `List` liefert `-1`, wenn das gesuchte Objekt nicht in der Liste ist und folglich auch keine Position vorhanden ist.
- Bei einer unbekannten Bytelänge einer MIDI-Datei (Typ `MidiFileFormat`) hat `getByteLength()` als Rückgabe `-1`.

Diese magischen Werte sollten vermieden werden, und daher kann auch der optionale Typ wieder erscheinen.

Als generischer Typ kann `Optional` beliebige Typen kapseln, und primitive Werte könnten in Wrapper verpackt werden. Allerdings bietet Java für drei primitive Typen spezielle `Optional`-Typen an: `OptionalInt`, `OptionalLong`, `OptionalDouble`:

Optional<T>	OptionalInt	OptionalLong	OptionalDouble
`static <T> Optional<T> empty()`	`static OptionalInt empty()`	`static OptionalLong empty()`	`static OptionalDouble empty()`
`T get()`	`int getAsInt()`	`long getAsLong()`	`double getAsDouble()`
`boolean isPresent()`			

Tabelle 1.13 Methodenvergleich zwischen den vier OptionalXXX-Klassen

17 Unter *http://download.java.net/jdk8/docs/api/constant-values.html* lassen sich alle Konstantendeklarationen einsehen.

Optional<T>	OptionalInt	OptionalLong	OptionalDouble
static <T> Optional<T> of(T value)	static OptionalInt of(int value)	static OptionalLong of(long value)	static OptionalDouble of(double value)
static <T> Optional<T> ofNullable(T value)	nicht übertragbar		
T orElse(T other)	int orElse(int other)	long orElse(long other)	double orElse(double other)
boolean equals(Object obj)			
int hashCode()			
String toString()			

Tabelle 1.13 Methodenvergleich zwischen den vier OptionalXXX-Klassen (Forts.)

Die `Optional`-Methode `ofNullable(…)` fällt in den primitiven `Optional`-Klassen natürlich raus. Die optionalen Typen für die drei primitiven Typen haben insgesamt weniger Methoden, und die obere Tabelle ist nicht ganz vollständig. Wir kommen im Rahmen der funktionalen Programmierung in Java noch auf die verbleibenden Methoden wie `isPresent(…)` zurück.

Best Practice

`OptionalXXX`-Typen eignen sich hervorragend als Rückgabetyp, sind als Parametertyp denkbar, doch wenig attraktiv für interne Attribute. Intern ist `null` eine akzeptable Wahl, der »Typ« ist schnell und speicherschonend.

Erstmal funktional mit Optional

Neben den vorgestellten Methoden wie `ofXXX(…)`, `isPresent()`, … gibt es weitere, die auf funktionale Schnittstellen zurückgreifen, die wir uns jetzt anschauen wollen:

```
final class java.lang.Optional<T>
```

- `void ifPresent(Consumer<? super T> consumer)`
  Repräsentiert das `Optional` einen Wert, rufe den `Consumer` mit diesem Wert auf, andernfalls mache nichts.
- `Optional<T> filter(Predicate<? super T> predicate)`
  Repräsentiert das `Optional` einen Wert und das Prädikat `predicate` auf dem Wert ist wahr, ist die Rückgabe das eigene `Optional`, sonst ist die Rückgabe `Optional.empty()`.

- `<U> Optional<U> map(Function<? super T,? extends U> mapper)`
  Repräsentiert das `Optional` einen Wert, dann wende die Funktion an, und verpacke das Ergebnis (wenn es ungleich `null` ist) wieder in ein `Optional`. War das `Optional` ohne Wert, dann ist die Rückgabe `Optional.empty()`, genauso, wenn die Funktion `null` liefert.
- `<U> Optional<U>flatMap(Function<? super T,Optional<U>> mapper)`
  Wie `map(…)`, nur dass die Funktion ein `Optional` statt eines direkten Werts gibt. Liefert die Funktion `mapper` ein leeres `Optional`, so ist das Ergebnis von `flatMap(…)` auch `Optional.empty()`.
- `T orElseGet(Supplier<? extends T> other)`
  Repräsentiert das `Optional` einen Wert, so liefere ihn, ist das `Optional` leer, so beziehe den Alternativwert aus dem `Supplier`.
- `<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)`
  Repräsentiert das `Optional` einen Wert, so liefere ihn, andernfalls hole mit `Supplier` das Ausnahme-Objekt, und löse es aus.

Beispiel für NullPointerException-sichere Kaskadierung von Aufrufen mit Optional

Die beiden `XXXmap(…)`-Methoden sind besonders interessant und ermöglichen einen ganz neuen Programmierstil. Warum, soll ein Beispiel zeigen.

Der folgende Zweizeiler gibt auf meinem System »MICROSOFT KERNELDEBUGGER-NETZWERKADAPTER« aus:

```
String s = NetworkInterface.getByIndex( 2 ).getDisplayName().toUpperCase();
System.out.println( s );
```

Allerdings ist der Programmcode alles andere als gut, denn `NetworkInterface.getByIndex(int)` kann `null` zurückgeben und `getDisplayName()` auch. Um ohne eine `NullPointerException` um die Klippen zu schiffen, müssen wir schreiben:

```
NetworkInterface networkInterface = NetworkInterface.getByIndex( 2 );
if ( networkInterface != null ) {
  String displayName = networkInterface.getDisplayName();
  if ( displayName != null )
    System.out.println( displayName.toUpperCase() );
}
```

Von der Eleganz des Zweizeilers ist nicht mehr viel geblieben. Integrieren wir `Optional` (was ja eigentlich ein toller Rückgabetyp für `getByIndex()` und `getDisplayName()` wäre):

```
Optional<NetworkInterface> networkInterface =
 Optional.ofNullable( NetworkInterface.getByIndex( 2 ) );
if ( networkInterface.isPresent() ) {
  Optional<String> name =
 Optional.ofNullable( networkInterface.get().getDisplayName() );
```

```
  if ( name.isPresent() )
    System.out.println( name.get().toUpperCase() );
}
```

Mit `Optional` wird es nicht sofort besser, doch statt `if` können wir einen Lambda-Ausdruck nehmen und bei `ifPresent(…)` einsetzen:

```
Optional<NetworkInterface> networkInterface =
 Optional.ofNullable( NetworkInterface.getByIndex( 2 ) );
networkInterface.ifPresent( ni -> {
  Optional<String> displayName = Optional.ofNullable( ni.getDisplayName() );
  displayName.ifPresent( name -> {
    System.out.println( name.toUpperCase() );
  } );
} );
```

Wenn wir nun die lokalen Variablen `networkInterface` und `displayName` entfernen, kommen wir aus bei:

```
Optional.ofNullable( NetworkInterface.getByIndex( 2 ) ).ifPresent( ni -> {
  Optional.ofNullable( ni.getDisplayName() ).ifPresent( name -> {
    System.out.println( name.toUpperCase() );
  } );
} );
```

Von der Struktur her ist das mit der `if`-Abfrage identisch und über die Einrückungen auch zu erkennen. Fallunterscheidungen mit `Optional` und `ifPresent(…)` umzuschreiben bringt also keinen Vorteil.

In Fallunterscheidungen zu denken hilft hier nicht weiter. Was wir uns bei `NetworkInterface.getByIndex( 2 ).getDisplayName().toUpperCase()` vor Augen halten müssen, ist eine Kette von Abbildungen. `NetworkInterface.getByIndex(int)` bildet auf `NetworkInterface` ab, `getDisplayName()` von `NetworkInterface` bildet auf `String` ab, und `toUpperCase()` bildet von einem `String` auf einen anderen `String` ab. Wir verketten also drei Abbildungen und müssten ausdrücken können: Wenn eine Abbildung fehlschlägt, dann höre mit der Abbildung auf. Und genau hier kommen `Optional` und `map(…)` ins Spiel. In Code:

```
Optional<String> s = Optional.ofNullable( NetworkInterface.getByIndex( 2 ) )
                              .map( ni -> ni.getDisplayName() )
                              .map( name -> name.toUpperCase() );
s.ifPresent( System.out::println );
```

Die Klasse `Optional` hilft uns bei zwei Dingen: Erstens wird `map(…)` beim Empfangen einer `null`-Referenz auf ein `Optional.empty()` abbilden, und zweitens ist das Verketten von leeren `Optional`s kein Problem, es passiert einfach nichts – `Optional.empty().map(…)` führt nichts aus, und die

Rückgabe ist einfach nur ein leeres `Optional`. Am Ende der Kette steht also nicht mehr `String` (wie am Anfang des Beispiels), sondern `Optional<String>`.

Umgeschrieben mit Methoden-Referenzen und weiter verkürzt ist der Code sehr gut lesbar und Null-Pointer-Exception-sicher:

```
Optional.ofNullable( NetworkInterface.getByIndex( 2 ) )
        .map( NetworkInterface::getDisplayName )
        .map( String::toUpperCase )
        .ifPresent( System.out::println );
```

Die Logik kommt ohne externe Fallunterscheidungen aus und arbeitet nur mit optionalen Abbildungen. Das ist ein schönes Beispiel für funktionale Programmierung.

Primitiv-Optionales

Die eigentliche `Optional`-Klasse ist generisch und kapselt jeden Referenztyp. Auch für die primitiven Typen `int`, `long` und `double` gibt es in drei spezielle Klassen `OptionalInt`, `OptionalLong`, `OptionalDouble` Methoden zur funktionalen Programmierung. Stellen wir die Methoden der vier `OptionalXXX`-Klassen gegenüber:

Optional<T>	OptionalInt	OptionalLong	OptionalDouble
`static <T> Optional<T> empty()`	`static OptionalInt empty()`	`static OptionalLong empty()`	`static OptionalDouble empty()`
`T get()`	`int getAsInt()`	`long getAsLong()`	`double getAsDouble()`
`boolean isPresent()`			
`static <T> Optional<T> of(T value)`	`static OptionalInt of(int value)`	`static OptionalLong of(long value)`	`static OptionalDouble of(double value)`
`static <T> Optional<T> ofNullable(T value)`	nicht übertragbar		
`T orElse(T other)`	`int orElse(int other)`	`long orElse(long other)`	`double orElse(double other)`
`boolean equals(Object obj)`			
`int hashCode()`			
`String toString()`			

Tabelle 1.14 Vergleich von Optional mit den primitiven OptionalXXX-Klassen

Optional<T>	OptionalInt	OptionalLong	OptionalDouble
void ifPresent(Consumer<? super T> consumer)	void ifPresent(IntConsumer consumer)	void ifPresent(LongConsumer consumer)	void ifPresent(DoubleConsumer consumer)
T orElseGet(Supplier<? extends T> other)	int orElseGet(IntSupplier other)	long orElseGet(LongSupplier other)	double orElseGet(DoubleSupplier other)
<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)	<X extends Throwable> int orElseThrow(Supplier<X> exceptionSupplier)	<X extends Throwable> long orElseThrow(Supplier<X> exceptionSupplier)	<X extends Throwable> double orElseThrow(Supplier<X> exceptionSupplier)
Optional<T> filter(Predicate<? super T> predicate)	nicht vorhanden		
<U> Optional<U> flatMap(Function <? super T,Optional<U>> mapper)			
<U> Optional<U> map(Function<? super T,? extends U> mapper)			

Tabelle 1.14 Vergleich von Optional mit den primitiven OptionalXXX-Klassen (Forts.)

1.2.9 Was ist jetzt so funktional?

Bisher hat dieser Abschnitt einen Großteil darauf verwendet, die Typen aus dem `java.util.function`-Paket vorzustellen, also die funktionalen Schnittstellen, mit denen Entwickler Abbildungen in Java ausdrücken können. Wenig war von funktionaler Programmierung und den Vorteilen die Rede, das holen wir jetzt nach.

Wiederverwertbarkeit

Zunächst einmal bieten Funktionen eine zusätzliche Ebene der Wiederverwertbarkeit von Code. Nehmen wir ein Prädikat wie

```
Predicate<Path> exists = path -> Files.exists( path );
```

Dieses `exists`-Prädikat ist relativ einfach, könnte aber natürlich komplexer sein. Der Punkt ist, dass diese Prädikate an allen möglichen Stellen wiederverwendet werden können, etwa zum Filtern in Listen, zum Löschen von Elementen aus Listen usw. Das Prädikat kann also als Funktion weitergereicht oder zu neuen Prädikaten verbunden werden, etwa zu:

```
Predicate<Path> exists    = path -> Files.exists( path );
Predicate<Path> directory = path -> Files.isDirectory( path );
Predicate<Path> existsAndDirectory = exists.and( directory );
```

Methoden wie `ifPresent(Predicate)` oder `removeIf(Predicate)` nehmen dann dieses Prädikat und führen Operationen durch. Diese kleinen Mini-Objekte lassen sich sehr gut testen, und das minimiert insgesamt Fehler im Code.

Während aktuelle Bibliotheken wenig davon Gebrauch machen, Typen wie `Supplier`, `Consumer`, `Function`, `Predicate` anzunehmen und zurückzugeben, wird sich dieses im Laufe der nächsten Jahre ändern.

Zustandslos, immutable

Bei der funktionalen Programmierung geht es darum, ohne externe Zustände auszukommen.

Definition

Funktionen heißen pur, wenn sie ohne einen Zustand auskommen und keine Seiteneffekte haben. `Math.max(3, 4)` ist eine pure Funktion, `System.out.println()` oder `Math.random()` sind es nicht. Einen aus puren Funktionen aufgebauten Ausdruck nennen wir *puren Ausdruck*. Er hat eine Eigenschaft, die sich in der Informatik *referenzielle Transparenz* nennt, dass nämlich das Ergebnis eines Ausdrucks an Stelle des Ausdrucks selbst gesetzt werden kann, ohne dass das Programm ein anderes Verhalten zeigt. Statt `Math.max(3, 4)` kann jederzeit 4 gesetzt werden, das Ergebnis wäre das gleiche. Ein Compiler kann bei referenzieller Transparenz diverse Optimierungen durchführen.

Pure funktionale Programmiersprachen basieren auf puren Funktionen, und auch in Java muss nicht jede Methode einen äußeren Zustand verändern. Allerdings sind es Java-Entwickler gewohnt, in Zuständen zu denken, und daran ist an sich nichts Falsches: Ein Textdokument im Speicher ist eben ein Objektgraph genauso wie eine grafische Anwendung mit Eingabefeldern. Worauf funktionale Programmierung aber abzielt, sind die Operationen auf den Datenstrukturen und Berechnungen, dass sie ohne Seiteneffekte sind.

Pure Funktionen ohne Zustand haben den Vorteil, dass sie

- beliebig oft ausgeführt werden können, ohne dass sich Systemzustände ändern,
- in beliebiger Reihenfolge ausgeführt werden können, ohne dass das Ergebnis ein anderes wird.

Diese Vorteile sind reizvoll unter dem Gesichtspunkt der Parallelisierung, denn die Prozessoren werden nicht wirklich schneller, aber wir haben mehr Prozessorkerne zur Verfügung. Pure Funktionen erlauben es Bibliotheken, Aufgaben wie Suchen und Filtern auf Kerne zu verteilen und so zu parallelisieren. Je weniger Zustand dabei im Spiel ist, desto besser, denn je weniger Zustand, desto weniger Synchronisation und Warteeffekte gibt es.

Aufpassen müssen Entwickler natürlich trotzdem, denn ein Lambda-Ausdruck muss nicht pur sein und kann Seiteneffekte haben. Daher ist es wichtig zu wissen, wann diese Lambda-Ausdrücke vielleicht nebenläufig sind und eine Synchronisation nötig ist.

Beispiel

Die Schnittstelle `Iterable` deklariert eine Methode `forEach(…)`, mit einem Parameter vom Typ einer funktionalen Schnittstelle. Hier ist ein Lambda-Ausdruck möglich. Es wäre natürlich grundlegend falsch, wenn dieser Lambda-Ausdruck selbst in die Sammlung eingreift:

```
List<Integer> ints = new ArrayList<>( Arrays.asList( 1, 99, 2 ) );
ints.forEach( v -> { System.out.println( ints + ", " + v); ints.set( v, 0 ); } );
```

Die Ausgabe ist weit von dem, was erwartet wurde, aber kein Wunder, wenn Lambda-Ausdrücke illegale Seiteneffekte hervorrufen:

```
[1, 99, 2], 1
[1, 0, 2], 0
[0, 0, 2], 2
```

Die Vermeidung von Zuständen gekoppelt an die Unveränderbarkeit von Werten (engl. *immutability*) erhöht das Verständnis des Programms, da Entwickler es schwer haben, im Kopf das System mit den ganzen Änderungen »nachzuspielen«, insbesondere wenn diese Änderungen noch nebenläufig sind. Das zeigt das vorangehende Beispiel recht gut; solche Systeme zu verstehen und zu debuggen ist schwer. Je weniger Seiteneffekte es gibt, desto einfacher ist das Programm zu verstehen. Zustände machen ein Programm komplex, nicht nur in nebenläufigen Umgebungen. Wenn die Methode pur ist, muss ein Entwickler nichts anderes tun, als den Code der Methode zu verstehen. Wenn die Methode von Zuständen des Objekts abhängt, muss ein Entwickler den Code der gesamten Klasse verstehen. Und wenn das Objekt von Zuständen im Gesamtprogramm abhängt, ufert das Ganze aus, denn dann ist noch viel mehr vom System zu verstehen.

1.2.10 Zum Weiterlesen

Funktional zu Programmieren ändert das grundlegend das Design von Java-Programmen: weg von Methoden mit Seiteneffekten hin zu kleinen Funktionen. Die Zukunft wird uns Muster und Best-Practices an die Hand geben, wie in Java entwickelt wird. Auch wird sich zeigen, ob weitere Konzepte der funktionalen Programmierung in Java bzw. die JVM einfließen werden. Bisher ist zum Beispiel Immutability kein Sprachkonstrukt, sondern durch die API gewährleistet, wenn es keine Setter oder Schreibzugriffe auf Variablen gibt; Reflection kann aber auch hier einen bösen Strich durch die Rechnung machen. Doch für Java 9 sind noch keine Planungen in dieser Richtung gemacht, das Gleiche gilt für Möglichkeiten wie Pattern Matching oder algebraische Datentypen (ADT) auf der Sprachseite oder Optimierung von Endrekursion (engl. *tail call optimization*) auf Seiten der JVM, etwas, das in anderen funktionalen Programmiersprachen immer hochgehalten wird.

Entwickler, die noch tiefer in die Denkweise funktionaler Programmierung eintauchen möchten, können sich mit puren funktionalen Programmiersprachen wie Haskell beschäftigen und müssen dort ohne Seiteneffekte auskommen. Etwas einfacher für Java-Programmierer ist die Sprachfamilie ML, die auch imperative Elemente wie `while`-Schleifen bietet. Für Java-Programmierer wirkt das meist fremd, die hippe Programmiersprache Scala vereinet objektorientierte und funktionale Programmierung nahezu perfekt.

1.3 Bibliotheksänderungen in Java 8

Dieser Band widmet sich ausführlich den Änderungen in Java 8. Die neue Stream-API ist Bestandteil von Kapitel 4, »Datenstrukturen und Algorithmen«, und die neue Date-Time-API wird in Kapitel 5, »Raum und Zeit«, vorgestellt.

1.4 JDK 8-HotSpot-JVM-Änderungen

Wenn es um Java geht, müssen wir immer unterscheiden, ob es um die Sprache selbst geht, um die Bibliotheken, um die JVM oder um die Implementierung von Oracle, die das JDK darstellt. Eine zentrale Neuerung in der Version 8 der HotSpot-JVM ist der Umgang mit der internen Repräsentation von Klassen und Metadaten; die Daten landen nicht mehr auf dem Java-Heap, sondern im nativen Speicher, der *Metaspace* heißt. Vorher gehörte ein `java.lang.OutOfMemoryError: PermGen space` bei großen Anwendungen zum Alltag.[18] Das Problem ist (im Prinzip) Vergangenheit, es gibt keinen Perm-Space mehr in der JDK 8-JVM; die Oracle JRockit und JVM von IBM hatten übrigens immer Metaspace. Der Metaspace-Speicherbedarf kann mit dem Schalter `MaxMetaspaceSize` begrenzt werden, ein Übertritt führt zur `java.lang.OutOfMemoryError: Metadata space`. Eine Begrenzung ist nicht verkehrt, das Ergebnis unter Java 7 und Java 8 aber das gleiche, ein `OutOfMemoryError`.

1.5 Auf Java 7/8-Syntax mit NetBeans und Eclipse migrieren

Wer eine große Codebasis auf Java 8 oder Java 7 migrieren möchte, steht vor dem Problem, wie das ohne großen manuellen Aufwand funktionieren kann. Als Erstes steht die Umsetzung auf die neuen Syntax-Möglichkeiten an. Das ist ein echter Mehrwert, und hier können moderne Entwicklungsumgebungen einiges als Vorarbeit leisten. Bei den Bibliotheken gibt es keine automatische Umsetzung von Alt nach Neu, hier ist leider Handarbeit angesagt.

Schauen wir uns an, welche Möglichkeiten die modernen IDEs bieten.

18 *http://stackoverflow.com/questions/88235/dealing-with-java-lang-outofmemoryerror-permgen-space-error*

1.5.1 Java 8-Syntax-Migration mit NetBeans

Die aktuelle NetBeans-IDE bringt einen Code-Transformator mit, der elegant die Codebasis durchsucht und Transformationen automatisch durchführt. Das Menü Refactor bietet den Unterpunkt Inspect and Transform..., der zu einem Dialog führt:

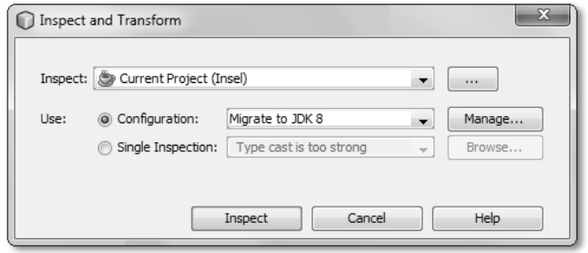

Abbildung 1.2 Transformationen in Java 8/Java 7 anstoßen

Unter Manage... öffnet sich ein weiterer Dialog, bei dem JDK Migration Support aktiviert ist und einige Änderungen voreingestellt sind; neu für Java 8 ist Convert to Lambda or Method References. Auch die Transformationen in die neue Java 7-Syntax sind voreingestellt, aktiviert werden kann: Can use Diamond, Convert to try-with-resources, Join catch sections using mulitcatch, Use specific catch und Use switch over Strings when possible.

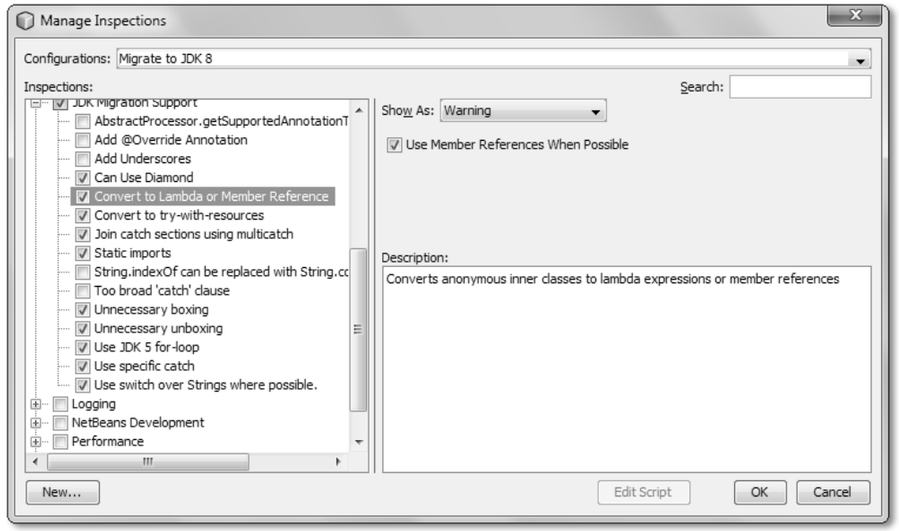

Abbildung 1.3 Einstellungsmöglichkeiten

Wer Programmcode neu mit NetBeans schreibt, wird gleich auf die neue Syntax hingewiesen. Vervollständigungen nutzen ganz natürlich zum Beispiel den Diamond-Operator.

1.5.2 Java 8-Syntax-Migration mit Eclipse

Seit der Version Eclipse 4.4 (bzw. Eclipse 4.3.2 mit Feature Pack für Java 8-Erweiterung) integriert die IDE einen Code-Transformator, der Klassenimplementierung von funktionalen Schnittstellen in Lambda-Ausdrücke verkürzen kann. Diese Transformationen sind über die Quick-Fixes/Quick-Assists möglich.

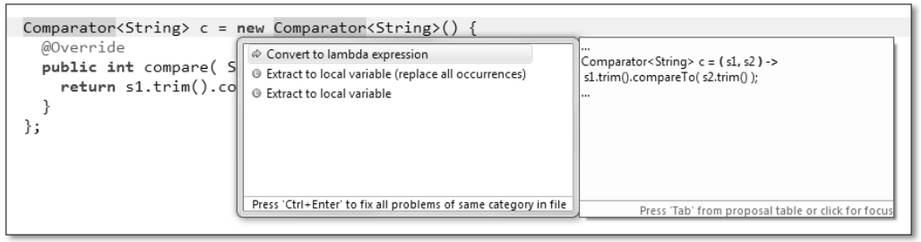

Abbildung 1.4 Konvertierung in Lambda-Ausdruck

Leider kann diese Umsetzung (bisher) nicht für ein ganzes Projekt erfolgen und keine umfangreiche Codebasis komplett und automatisch auf die neuen Lambda-Ausdrücke gebracht werden.

1.5.3 File-Klassen auf NIO.2 umstellen

Um sich von der Klasse `java.io.File` zu lösen und in Richtung `Path` zu migrieren, ist Handarbeit angesagt. Zunächst gilt es, alle Stellen zu finden, die im Workspace oder Projekt die Klasse `File` referenzieren. Am einfachsten ist es, eine Anweisung wie `File f;` in den Code zu setzen, dann zum Beispiel in Eclipse Strg + ⇧ + G zu aktivieren. Es folgt eine Liste aller Vorkommen. Diese Liste muss nun abgearbeitet werden und der Code auf NIO.2 konvertiert werden. Nicht immer ist es so einfach

```
Scanner s = new Scanner( new File(dateiname) );
```

in

```
Scanner s = new Scanner( Paths.get(dateiname) );
```

umzusetzen oder

```
new File( dateiname ).delete();
```

in

```
Files.delete( Paths.get( dateiname ) );
```

Eine andere Stelle, an der Konvertierungen möglich sind, betreffen `FileReader` und `FileWriter`. Diese Klassen sind gefährlich, weil sie standardmäßig die im System voreingestellte Kodierung verwenden. Eine Kodierung (wie UTF-8) im Konstruktor explizit vorzugeben ist jedoch nicht

möglich. NIO.2 bietet eine bessere Methode, um an einen `Reader`/`Writer` aus einer Datei zu kommen, wobei in Java 8 Entwickler auch nicht gezwungen werden, diese Kodierung immer sichtbar anzugeben, denn das ist optional, ohne Angabe ist UTF-8 voreingestellt.

Deklaration	Klassisch	Mit NIO.2
`Reader r =`	`new FileReader(f);`	`Files.newBufferedReader(Paths.get(f), StandardCharsets.ISO_8859_1);`
`Writer w =`	`new FileWriter(f);`	`Files.newBufferedWriter(Paths.get(f), StandardCharsets.ISO_8859_1);`

Tabelle 1.15 Beziehen eines Dateistroms klassisch und mit NIO.2

Der Quellcode wird erst einmal länger, doch der Gewinn ist, dass die Kodierung nicht verschwindet und auch die Ein-/Ausgabe gleich gepuffert ist. Freunde der statischen Import-Anweisung komprimieren weiterhin zu Anweisungen wie:

```
Reader r = newBufferedReader( get( f ), ISO_8859_1 );
```

Es ist auf jeden Fall manuelle Arbeit angesagt. Eigene Blöcke, etwa zum Schreiben in Dateien, können komplett zusammengestrichen werden auf einfache Methodenaufrufe wie:

```
Files.write( Paths.get( f ), bytes );
byte[] bytes = Files.readAllBytes( Paths.get( f ) );
Files.write( Paths.get( f ), Collections.singleton( "Zeile 1" ),
                             StandardCharsets.ISO_8859_1  );
List<String> lines = Files.readAllLines( Paths.get( f ),
                                         StandardCharsets.ISO_8859_1 );
```

1.6 Zum Weiterlesen

Alle neuen Typen und Eigenschaften von Java 8 listet die Webseite *http://www.tutego.de/java/java-8-opendjk-8-java-se-8.html* auf. Welche Quick-Fixes Eclipse für Java 7 bietet, listet *http://www.eclipse.org/jdt/ui/r3_8/Java7news/whats-new-java-7.html* auf. Eine zusammenfassende Liste der Eclipse-Neuerungen für Java 8 war zum Zeitpunkt der Drucklegung noch nicht verfügbar.

Kapitel 8

Die eXtensible Markup Language (XML)

»Ich bin überall in diesem Land gewesen und habe mit den besten Leuten gesprochen. Ich kann Ihnen versichern, dass Datenverarbeitung ein Tick ist, der sich nächstes Jahr erledigt hat.«
– Editor für Computerbücher bei Prentice Hall, 1957

8.1 Auszeichnungssprachen

Auszeichnungssprachen dienen der strukturierten Gliederung von Texten und Daten. Ein Text besteht zum Beispiel aus Überschriften, Fußnoten und Absätzen, eine Vektorgrafik dagegen aus einzelnen Grafikelementen wie Linien und Textfeldern. Auszeichnungssprachen liegt die Idee zugrunde, besondere Bausteine durch Auszeichnung hervorzuheben. Ein Text könnte etwa so beschrieben sein:

```
<Überschrift>
Mein Buch
<Ende Überschrift>
Hui ist das <fett>toll<Ende fett>.
```

Als Leser eines Buchs erkennen wir optisch eine Überschrift an der Schriftart. Ein Computer hat damit aber seine Schwierigkeiten. Wir wollen auch dem Rechner die Fähigkeit verleihen, diese Struktur zu erkennen.

HTML ist die erste populäre Auszeichnungssprache, die Auszeichnungselemente (engl. *tags*) wie `<b>fett</b>` benutzt, um bestimmte Eigenschaften von Elementen zu kennzeichnen. Damit wurde eine Visualisierung verbunden, etwa bei einer Überschrift fett und mit großer Schrift. Leider werden Auszeichnungssprachen wie HTML auch dazu benutzt, Formatierungseffekte zu erzielen. Beispielsweise werden Überschriften richtigerweise mit dem Überschriften-Tag ausgezeichnet. Wenn an anderer Stelle eine Textstelle fett und groß sein soll, wird diese aber auch oft mit dem Überschriften-Tag markiert, obwohl sie keine Überschrift ist.

8.1.1 Die Standard Generalized Markup Language (SGML)

Das Beispiel der Überschrift in einem Buch veranschaulicht die Idee, Bausteine mit Typen in Verbindung zu bringen. Der allgemeine Aufbau mit diesen Auszeichnungselementen ließe sich dann für beliebige hierarchische Dokumente nutzen. Die Definition einer Auszeichnungssprache (Metasprache) ist daher auch nicht weiter schwierig. Schon Mitte der 1980er Jahre wurde als

ISO-Standard die *Standard Generalized Markup Language* (SGML)[1] definiert, die die Basis für beliebige Auszeichnungssprachen ist. Ab der Version 2.0 ist auch HTML als SGML-Anwendung definiert. Leider kam mit den vielen Freiheiten und der hohen Flexibilität eine große und aufwändige Deklaration der Anwendungen hinzu. Ein SGML-Dokument musste einen ganz bestimmten Aufbau besitzen. SGML-Dateien waren daher etwas unflexibel, weil die Struktur genau eingehalten werden musste. Für HTML-Dateien wäre das schlecht, weil die Browser-Konkurrenten produktspezifische Tags definieren, die auf den Browser des jeweiligen Herstellers beschränkt bleiben. So interpretiert der Internet Explorer zum Beispiel das Tag `<blink>blinkend </blink>` nicht. Tags, die ein Browser nicht kennt, überliest er einfach.

8.1.2 Extensible Markup Language (XML)

Für reine Internetseiten hat sich HTML etabliert, aber für andere Anwendungen wie Datenbanken oder Rechnungen ist HTML nicht geeignet. Für SGML sprechen die Korrektheit und Leistungsfähigkeit – dagegen sprechen die Komplexität und die Notwendigkeit, eine Beschreibung für die Struktur angeben zu müssen. Daher setzte sich das W3C zusammen, um eine neue Auszeichnungssprache zu entwickeln, die einerseits so flexibel wie SGML, andererseits aber so einfach zu nutzen und zu implementieren ist wie HTML. Das Ergebnis war die *eXtensible Markup Language* (XML). Diese Auszeichnungssprache ist für Compiler einfach zu verarbeiten, da es genaue Vorgaben dafür gibt, wann ein Dokument in Ordnung ist.

XML ist nicht nur der Standard zur Beschreibung von Daten, denn oft verbinden sich mit diesem Ausdruck eine oder mehrere Technologien, die mit der Beschreibungssprache im Zusammenhang stehen. Und: Ohne XML kein WiX[2]! Die wichtigsten Technologien zur Verarbeitung von XML in Java werden hier kurz vorgestellt. Eine ausführliche Beschreibung mit allen Nachbartechnologien finden Sie bei Interesse auf den Webseiten des W3C unter *http://www.w3c.org/*.

8.2 Eigenschaften von XML-Dokumenten

8.2.1 Elemente und Attribute

Der Inhalt eines XML-Dokuments besteht aus strukturierten *Elementen*, die hierarchisch geschachtelt sind. Dazwischen befindet sich der Inhalt, der aus weiteren Elementen (daher »hierarchisch«) und reinem Text bestehen kann. Die Elemente können *Attribute* enthalten, die zusätzliche Informationen in einem Element ablegen:

1 Der Vorgänger von SGML war GML; hier standen die Buchstaben (sicherlich inoffiziell) für Charles Goldfarb, Edward Mosher und Raymond Lorie, die bei IBM in den 1960er Jahren diese Dokumentenbeschreibungssprache geschaffen hatten.

2 Windows Installer XML – Definition von Auslieferungspaketen für Microsoft Windows

Listing 8.1 party.xml

```
<?xml version="1.0"?>
<party datum="31.12.2012">
  <gast name="Albert Angsthase">
    <getraenk>Wein</getraenk>
    <getraenk>Bier</getraenk>
    <zustand ledig="true" nuechtern="false"/>
  </gast>
</party>
```

Die Groß- und Kleinschreibung der Namen für Elemente und Attribute ist für die Unterscheidung wichtig. Ein Attribut besteht aus einem Attributnamen und einem Wert. Der Attributwert steht immer in einfachen oder doppelten Anführungszeichen, und das Gleichheitszeichen weist dem Attributnamen den Wert zu.

Verwendung von Tags

Gemäß der *Reference Concrete Syntax* geben Elemente in spitzen Klammern die Tags an. Elemente existieren in zwei Varianten: Falls das Element einen Wert einschließt, besteht es aus einem Anfangs-Tag und einem End-Tag.

Element = öffnendes Tag + Inhalt + schließendes Tag

Das Anfangs-Tag gibt den Namen des Tags vor und enthält die Attribute. Das End-Tag hat den gleichen Namen wie das Anfangs-Tag und wird durch einen Schrägstrich nach der ersten Klammer gekennzeichnet. Zwischen dem Anfangs- und dem End-Tag befindet sich der Inhalt des Elements.

Beispiel

Das Element `<getraenk>` mit dem Wert `Wein`:

```
<getraenk>Wein</getraenk>
```

Ein Element, das keine Inhalte einschließt, besteht aus nur einem Tag mit einem Schrägstrich vor der schließenden spitzen Klammer. Diese Tags haben entweder Attribute als Inhalt, oder das Auftreten des Tags ist Bestandteil des Inhalts.

Beispiel

Das Element `<zustand>` mit den Attributen `ledig` und `nuechtern`:

```
<zustand ledig="true" nuechtern="false" />
```

Bedeutung der Tags

Durch die freie Namensvergabe in XML-Dokumenten ist eine formatierte Darstellung eines Dokuments nicht möglich. Anders als bei HTML gibt es keine festgelegte Menge von Tags, die den Inhalt nach bestimmten Kriterien formatieren. Falls das XML-Dokument in einem Browser dargestellt werden soll, sind zusätzliche Beschreibungen in Form von Formatvorlagen (Stylesheets) für die Darstellung in HTML notwendig.

Wohlgeformt

Ein XML-Dokument muss einige Bedingungen erfüllen, damit es *wohlgeformt* ist. Wenn es nicht wohlgeformt ist, ist es auch kein XML-Dokument. Damit ein XML-Dokument wohlgeformt ist, muss jedes Element aus einem Anfangs- und einem End-Tag oder nur aus einem abgeschlossenen Tag bestehen. Hierarchische Elemente müssen in umgekehrter Reihenfolge ihrer Öffnung wieder geschlossen werden. Die Anordnung der öffnenden und schließenden Tags legt die Struktur eines XML-Dokuments fest. Jedes XML-Dokument muss ein Wurzelelement enthalten, das alle anderen Elemente einschließt.

Beispiel

Das Wurzelelement heißt `<party>` und schließt das Element `<gast>` ein:

```
<party datum="31.12.11">
  <gast name="Albert Angsthase"></gast>
</party>
```

Spezielle Zeichen in XML (Entitäten)

Wir müssen darauf achten, dass einige Zeichen in XML bestimmte Bedeutungen haben. Dazu gehören `&`, `<`, `>`, `"` und `'`. Sie werden im Text durch spezielle Abkürzungen, die *Entitäten*, abgebildet. Dies sind für die oben genannten Zeichen `&`, `<`, `>`, `&quot;` und `&apos;`. Diese Entitäten für die Sonderzeichen sind als einzige durch den Standard festgelegt.

<!-- Kommentare -->

XML-Dokumente können auch Kommentare enthalten. Diese werden beim Auswerten der Daten übergangen. Kommentare verbessern die Qualität des XML-Dokuments für den Benutzer wesentlich. Sie können an jeder Stelle des Dokuments verwendet werden, nur nicht innerhalb der Tags. Kommentare haben die Form:

```
<!-- Text des Kommentars -->
```

Der beste Kommentar eines XML-Dokuments ist die sinnvolle Gliederung des Dokuments und die Wahl selbsterklärender Namen für Tags und Attribute.

Kopfdefinition

Die Wohlgeformtheit muss mindestens erfüllt sein. Zusätzlich dürfen andere Elemente eingebaut werden. Dazu gehört etwa eine Kopfdefinition, die beispielsweise

```
<?xml version="1.0"?>
```

lauten kann. Diese Kopfdefinition lässt sich durch Attribute erweitern. In diesem Beispiel werden die verwendete XML-Version und die Zeichenkodierung angegeben:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Wenn eine XML-Deklaration vorhanden ist, muss sie ganz am Anfang des Dokuments stehen. Dort lässt sich im Prinzip die benutzte Zeichenkodierung definieren, wenn sie nicht automatisch UTF-8 oder UTF-16 ist. Automatisch kann jedes beliebige Unicode-Zeichen unabhängig von der Kodierung über das Kürzel `&#xABCD;` (`A`, `B`, `C`, `D` stehen für Hexadezimalzeichen) dargestellt werden.

Hinweis

Java und andere XML-Parser nehmen standardmäßig die Zeichenkodierung UTF-8 an. Es ist daher eine gute Idee, grundsätzlich alle XML-Dokumente in UTF-8 abzulegen.

8.2.2 Beschreibungssprache für den Aufbau von XML-Dokumenten

Im Gegensatz zu HTML ist bei XML die Menge der Tags und deren Kombination nicht festgelegt. Für jede Anwendung können Entwickler beliebige Tags definieren und verwenden. Um aber überprüfen zu können, ob eine XML-Datei für eine bestimmte Anwendung die richtige Form hat, wird eine formale Beschreibung dieser Struktur benötigt. Diese formale Struktur ist in einem bestimmten Format beschrieben, wobei zwei Formate populär sind: das *XML Schema* und die *Document Type Definition* (DTD). Sie legen fest, welche Tags zwingend vorgeschrieben sind, welche Art Inhalt diese Elemente haben, wie Tags miteinander verschachtelt sind und welche Attribute ein Element besitzen darf. Hält sich ein XML-Dokument an die Definition, ist es *gültig* (engl. *valid*).

Mittlerweile gibt es eine große Anzahl von Beschreibungen in Form von Schemas und DTDs, die Gültigkeiten für die verschiedensten Daten definieren. Einige DTDs sind unter *http://tutego.de/go/xmlapplications* aufgeführt. Um einen Datenaustausch für eine bestimmte Anwendung zu gewährleisten, ist eine eindeutige Beschreibung unerlässlich. Es wäre problematisch, wenn die Unternehmen unter der Struktur einer Rechnung immer etwas anderes verstünden.

Document Type Definition (DTD)

Für die folgende XML-Datei entwickeln wir eine DTD zur Beschreibung der Struktur:

Listing 8.2 party.xml

```
<?xml version="1.0" ?>
<party datum="31.12.2012">
    <gast name="Albert Angsthase">
        <getraenk>Wein</getraenk>
        <getraenk>Bier</getraenk>
        <zustand ledig="true" nuechtern="false"/>
    </gast>
    <gast name="Martina Mutig">
        <getraenk>Apfelsaft</getraenk>
        <zustand ledig="true" nuechtern="true"/>
    </gast>
    <gast name="Zacharias Zottelig"></gast>
</party>
```

Für diese XML-Datei legen wir die Struktur fest und beschreiben sie in einer DTD. Dazu sammeln wir zuerst die Daten, die in dieser XML-Datei stehen:

Elementname	Attribute	Untergeordnete Elemente	Aufgabe
party	datum Datum der Party	gast	Wurzelelement mit dem Datum der Party als Attribut
gast	name Name des Gastes	getraenk und zustand	die Gäste der Party; Name des Gastes als Attribut
getraenk			Getränk des Gastes als Text
zustand	ledig und nuechtern		Familienstand und Zustand als Attribute

Tabelle 8.1 Struktur der Beispiel-XML-Datei

Elementbeschreibung

Die Beschreibung der Struktur eines Elements besteht aus dem Elementnamen und dem Typ. Sie kann auch aus einem oder mehreren untergeordneten Elementen in Klammern bestehen. Der Typ legt die Art der Daten in dem Element fest. Mögliche Typen sind etwa PCDATA (*Parsed Character Data*) für einfachen Text oder ANY für beliebige Daten.

Untergeordnete Elemente werden als Liste der Elementnamen angegeben. Die Namen sind durch ein Komma getrennt. Falls verschiedene Elemente oder Datentypen alternativ vorkommen können, werden diese ebenfalls in Klammern angegeben und mit dem Oder-Operator (|) verknüpft. Hinter jedem Element und hinter der Liste von Elementen legt ein Operator fest, wie häufig das Element oder die Folgen von Elementen erscheinen müssen. Falls kein Operator angegeben ist, muss das Element oder die Elementliste genau einmal erscheinen. Folgende Operatoren stehen zur Verfügung:

Operator	Wie oft erscheint das Element?
?	einmal oder gar nicht
+	mindestens einmal
*	keinmal, einmal oder beliebig oft

Tabelle 8.2 DTD-Operatoren für Wiederholungen

Beispiel

Das Element <party> erlaubt beliebig viele Unterelemente vom Typ <gast>:

```
<!ELEMENT party (gast)*>
```

Drückt aus, dass auf einer Party beliebig viele Gäste erscheinen können.

Attributbeschreibung

Die Beschreibung der Attribute sieht sehr ähnlich aus. Sie besteht aus dem Element, den Attributnamen, den Datentypen der Attribute und einem Modifizierer. In einem Attribut können als Werte keine Elemente angegeben werden, sondern nur Datentypen wie CDATA (*Character Data*). Der Modifizierer legt fest, ob ein Attribut zwingend vorgeschrieben ist oder nicht. Folgende Modifizierer stehen zur Verfügung:

Modifizierer	Erläuterung
#IMPLIED	Muss nicht vorkommen.
#REQUIRED	Muss auf jeden Fall vorkommen.
#FIXED [Wert]	Wert wird gesetzt und kann nicht verändert werden.

Tabelle 8.3 Attribut-Modifizierer

Beispiel

Das Attribut datum für das Element <party>:

```
<!ATTLIST party datum CDATA #REQUIRED>
```

Der Wert des Attributs datum ist Text und muss angegeben sein (festgelegt durch den Modifizierer #REQUIRED).

Kümmern wir uns um die Beschreibung eines Gastes, der einen Namen und einen Zustand hat:

```
<!ELEMENT gast (getraenk*, zustand?)>
<!ATTLIST gast name CDATA #REQUIRED>
```

Das Element hat als Attribut `name` und die Unterelemente `<getraenk>` und `<zustand>`. Ein Gast kann kein Getränk, ein Getränk oder viele einnehmen. Die Attribute des Elements `<zustand>` müssen genau einmal oder gar nicht vorkommen.

Das Element `<getraenk>` hat keine Unterelemente, aber einen Text, der das Getränk beschreibt:

```
<!ELEMENT getraenk (#PCDATA)>
```

Das Element `<zustand>` hat keinen Text und keine Unterelemente, aber die Attribute `ledig` und `nuechtern`, die mit Text gefüllt sind. Die Attribute müssen nicht unbedingt angegeben werden (Modifizierer `#IMPLIED`).

```
<!ELEMENT zustand EMPTY>
<!ATTLIST zustand ledig CDATA #IMPLIED
   nuechtern CDATA #IMPLIED>
```

Bezugnahme auf eine DTD

Falls die DTD in einer speziellen Datei steht, wird im Kopf der XML-Datei angegeben, wo die DTD für dieses XML-Dokument steht:

```
<!DOCTYPE party SYSTEM "dtd\partyfiles\party.dtd">
```

Hinter `DOCTYPE` steht das Wurzelelement der zu beschreibenden XML-Datei. Hinter `SYSTEM` steht der URI mit der Adresse der DTD-Datei. Die DTD selbst kann in einer eigenen Datei stehen oder Bestandteil der XML-Datei sein.

Die vollständige DTD zu dem Party-Beispiel sieht folgendermaßen aus:

Listing 8.3 party.dtd

```
<!ELEMENT party (gast)*>
<!ATTLIST party datum CDATA #REQUIRED>
<!ELEMENT gast (getraenk*, zustand?)>
<!ATTLIST gast name CDATA #REQUIRED>
<!ELEMENT getraenk (#PCDATA)>
<!ELEMENT zustand EMPTY>
<!ATTLIST zustand ledig CDATA #IMPLIED nuechtern CDATA #IMPLIED>
```

Diese DTD definiert somit die Struktur aller XML-Dateien, die die Party beschreiben.

8.2.3 Schema – die moderne Alternative zu DTD

Ein anderes Verfahren, um die Struktur von XML-Dateien zu beschreiben, ist das *Schema*. Es ermöglicht eine Strukturbeschreibung wie eine DTD – nur in Form einer XML-Datei. Das vereinfacht das Parsen der Schema-Datei, da die Strukturbeschreibung und die Daten vom gleichen Dateityp sind. Ein Schema beschreibt im Gegensatz zu einer DTD die Datentypen der Elemente und Attribute einer XML-Datei viel detaillierter. Die üblichen Datentypen wie `string`, `integer` und `double` der gängigen Programmiersprachen sind bereits vorhanden. Weitere Datentypen wie `date` und `duration` existieren ebenfalls. Zusätzlich ist es möglich, eigene Datentypen zu definieren. Mit einem Schema kann weiterhin festgelegt werden, ob ein Element wie eine Ganzzahl in einem speziellen Wertebereich liegt oder ein String auf einen regulären Ausdruck passt. Die Vorteile sind eine genauere Beschreibung der Daten, die in einer XML-Datei dargestellt werden. Das macht aber auch die Strukturbeschreibung aufwändiger als mit einer DTD. Durch die detaillierte Beschreibung der XML-Struktur ist jedoch der Mehraufwand gerechtfertigt.

Party-Schema

Hier ist ein Beispiel für ein Schema, das die Struktur der Datei *party.xml* beschreibt:

Listing 8.4 party.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <xsd:element name="party" type="partyType" />

   <xsd:complexType name="partyType">
      <xsd:sequence>
         <xsd:element name="gast" type="gastType" />
      </xsd:sequence>
      <xsd:attribute name="datum" type="datumType" />
   </xsd:complexType>

   <xsd:complexType name="gastType">
      <xsd:sequence>
         <xsd:element name="getraenk" type="xsd:string" />
         <xsd:element name="zustand" type="zustandType" />
      </xsd:sequence>
   </xsd:complexType>

   <xsd:simpleType name="datumType">
      <xsd:restriction base="xsd:string">
         <xsd:pattern value="[0-3][0-9].[0-1][0-9].[0-9]{4}" />
      </xsd:restriction>
   </xsd:simpleType>
```

```
   <xsd:complexType name="zustandType">
      <xsd:complexContent>
         <xsd:restriction base="xsd:anyType">
            <xsd:attribute name="nuechtern" type="xsd:boolean" />
            <xsd:attribute name="ledig" type="xsd:boolean" />
         </xsd:restriction>
      </xsd:complexContent>
   </xsd:complexType>

</xsd:schema>
```

In diesem Beispiel werden die Typen `string` (für die Beschreibung des Elements `<getraenk>`) und `boolean` (für die Beschreibung des Elements `<ledig>`) verwendet. Die Typen `gastType` und `datumType` sind selbst definierte Typen. Ein sehr einfacher regulärer Ausdruck beschreibt die Form eines Datums. Ein Datum besteht aus drei Gruppen zu je zwei Ziffern, die durch Punkte getrennt werden. Die erste Ziffer der ersten Zifferngruppe muss aus dem Zahlenbereich 0 bis 3 stammen.

In der Schema-Datei basieren die Typen `datumType` und `zustandType` auf vorhandenen Schema-Typen, um diese einzuschränken. So schränkt `datumType` den Typ `string` auf die gewünschte Form eines Datums ein, und `zustandType` schränkt den `anyType` auf die beiden Attribute `nuechtern` und `ledig` ein. Die Schreibweise erzeugt einen neuen Typ, der keinen Text als Inhalt enthält, sondern nur die beiden Attribute `nuechtern` und `ledig` erlaubt. Der Wert der beiden Attribute ist ein Wahrheitswert.

Simple und komplexe Typen

Ein XML-Schema unterscheidet zwischen simplen und komplexen Typen. Simple Typen sind alle Typen, die keine Unterelemente und keine Attribute haben, sondern nur textbasierten Inhalt.

Beispiel

Das Element `<getraenk>` besteht nur aus einer Zeichenkette:

```
<xsd:element name="getraenk" type="xsd:string" />
```

Komplexe Typen können neben textbasiertem Inhalt auch Unterelemente und Attribute inkludieren.

Beispiel

Das Element `<gast>` hat den Typ `gastType` und die Unterelemente `<getraenk>` und `<zustand>`:

```
<xsd:element name="gast" type="gastType" />
<xsd:complexType name="gastType">
```



```
   <xsd:sequence>
     <xsd:element name="getraenk" type="xsd:string" />
     <xsd:element name="zustand" type="zustandType" />
   </xsd:sequence>
</xsd:complexType>
```

Simple und komplexe Typen können andere Typen einschränken. Komplexe Typen können zusätzlich andere Typen erweitern. Beim Erweitern ist es möglich, mehrere Typen miteinander zu kombinieren, um einen neuen Typ mit Eigenschaften verschiedener Typen zu erschaffen.

Das vorige Beispiel kann nur einen kleinen Einblick in die Möglichkeiten von XML-Schemas geben. Eine umfangreiche Dokumentation ist unter der URL *http://www.w3.org/XML/Schema* vorhanden. Dort gibt es drei verschiedene Dokumentationen zum Schema:

- **Schema Part0 Primer**: gut lesbares Tutorial mit vielen Beispielen
- **Schema Part1 Structures**: genaue Beschreibung der Struktur einer Schema-Datei
- **Schema Part2 Datatypes**: Beschreibung der Datentypen, die in XML-Schemas verwendet werden

Der erste Teil bietet eine grundlegende Einführung mit vielen Beispielen. Die beiden anderen Teile dienen als Referenzen für spezielle Fragestellungen.

8.2.4 Namensraum (Namespace)

Das Konzept des *Namensraums* ist besonders wichtig, wenn

- XML-Daten nicht nur lokal mit einer Anwendung benutzt werden,
- Daten ausgetauscht oder
- XML-Dateien kombiniert werden.

Eine Überschneidung der Namen der Tags, die in den einzelnen XML-Dateien verwendet werden, lässt sich nicht verhindern. Daher ist es möglich, einer XML-Datei einen Namensraum oder mehrere Namensräume zuzuordnen.

Der Namensraum ist eine Verknüpfung zwischen einem Präfix, das vor den Elementnamen steht, und einem URI. Ein Namensraum wird als Attribut an ein Element (typischerweise das Wurzelelement) gebunden und kann dann von allen Elementen verwendet werden. Das Attribut hat die Form:

```
xmlns:Präfix="URI"
```

Dem Element, das den Namensraum deklariert, wird ein Präfix vorangestellt. Es hat die Form:

```
<Präfix:lokaler Name xmlns:Präfix="URI">
```

Das Präfix ist ein frei wählbares Kürzel, das den Namensraum benennt. Dieses Kürzel wird dem Namen der Elemente, die zu diesem Namensraum gehören, vorangestellt. Der Name eines Elements des Namensraums `Präfix` hat die Form:

```
<Präfix:lokaler Name>...</Präfix:lokaler Name>
```

Angenommen, wir möchten für unsere Party das Namensraum-Präfix `geburtstag` verwenden. Der URI für diesen Namensraum ist *http://www.geburtstag.de*. Der Namensraum wird in dem Wurzelelement `party` deklariert. Das Präfix wird jedem Element zugeordnet:

```
<geburtstag:party xmlns:geburtstag="http://www.geburtstag.de"
                  geburtstag:datum="31.12.2012">
  <geburtstag:gast geburtstag:name="Albert Angsthase">
  </geburtstag:gast>
</geburtstag:party>
```

Eine weitere wichtige Anwendung von Namensräumen ist es, Tags bestimmter Technologien zu kennzeichnen. Für die XML-Technologien, etwa für Schemas, werden feste Namensräume vergeben.

Beispiel

Namensraumdefinition für ein XML-Schema:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Eine Anwendung, die XML-Dateien verarbeitet, kann anhand des Namensraums erkennen, welche Technologie verwendet wird. Dabei ist nicht das Präfix, sondern der URI für die Identifikation des Namensraums entscheidend. Für XML-Dateien, die eine Strukturbeschreibung in Form eines Schemas definieren, ist es üblich, das Präfix `xsd` zu verwenden. Es ist aber jedes andere Präfix möglich, wenn der URI auf die Adresse *http://www.w3.org/2001/XMLSchema* verweist. Diese Adresse muss nicht unbedingt existieren, und eine Anwendung kann auch nicht erwarten, dass sich hinter dieser Adresse eine konkrete HTML-Seite verbirgt. Der URI dient nur der Identifikation des Namensraums für eine XML-Datei.

8.2.5 XML-Applikationen *

Eine *XML-Applikation* ist eine festgelegte Auswahl von XML-Elementen und einem Namensraum. XHTML ist eine XML-Applikation, bei der die XML-Elemente die HTML-Elemente zur Beschreibung von Webseiten sind. Durch die Beschränkung auf eine bestimmte Menge von Elementen ist es möglich, diese XML-Dateien für bestimmte Anwendungen zu nutzen. Der Namensraum legt fest, zu welcher Applikation die einzelnen XML-Elemente gehören. Dadurch können verschiedene XML-Applikationen miteinander kombiniert werden.

Eine bekannte XML-Applikation ist XHTML. Unterschiedliche DTDs beschreiben die Menge möglicher Tags. Für XHTML 1.0 sind es folgende:

- *XHTML1-strict.dtd:* minimale Menge von HTML-Tags
- *XHTML1-transitional.dtd:* die gängigsten HTML-Tags
- *XHTML1-frameset.dtd:* HTML-Tags zur Beschreibung von Frames

Der Standard XHTML 1.1 geht noch einen Schritt weiter und bietet modulare DTDs an. Hier kann sehr genau differenziert werden, welche HTML-Tags für die eigene XML-Applikation gültig sind. Dadurch ist es sehr einfach möglich, XHTML-Elemente mit eigenen XML-Elementen zu kombinieren. Durch die Verwendung von Namensräumen können die XHTML- und die XML-Tags zur Datenbeschreibung unterschieden werden.

8.3 Die Java-APIs für XML

Für XML-basierte Daten gibt es vier Verarbeitungstypen:

- **DOM-orientierte APIs** (repräsentieren den XML-Baum im Speicher): *W3C-DOM, JDOM, dom4j, XOM ...*
- **Pull-API** (wie ein Tokenizer wird über die Elemente gegangen): Dazu gehören *XPP* (XML Pull Parser), wie sie der StAX-Standard definiert.
- **Push-API** (nach dem Callback-Prinzip ruft der Parser Methoden auf und meldet Elementvorkommen): *SAX* (Simple API for XML) ist der populäre Repräsentant.
- **Mapping-API** (der Nutzer arbeitet überhaupt nicht mit den Rohdaten einer XML-Datei, sondern bekommt die XML-Datei auf ein Java-Objekt umgekehrt abgebildet): *JAXB, XStream, Castor...*

Während DOM das gesamte Dokument in einer internen Struktur einliest und bereitstellt, verfolgt SAX einen ereignisorientierten Ansatz. Das Dokument wird in Stücken geladen, und immer dann, wenn ein angemeldetes Element beim Parser vorbeikommt, meldet er dies in Form eines Ereignisses, das für die Verarbeitung abgefangen werden kann.

Klassische Anwendungen für SAX und StAX sind:

- die Suche nach bestimmten Inhalten
- das Einlesen von XML-Dateien, um eine eigene Datenstruktur aufzubauen

Für einige Anwendungen ist es erforderlich, die gesamte XML-Struktur im Speicher zu verarbeiten. Für diese Fälle ist eine Struktur, wie DOM sie bietet, notwendig:

- Sortierung der Struktur oder einer Teilstruktur der XML-Datei
- Auflösen von Referenzen zwischen einzelnen XML-Elementen
- interaktives Arbeiten mit der XML-Datei

Ob ein eigenes Programm DOM oder StAX einsetzt, ist von Fall zu Fall unterschiedlich. In manchen Fällen ist dies auch Geschmackssache, doch unterscheidet sich das Programmiermodell, sodass eine Umstellung nicht so angenehm ist.

8.3.1 Das Document Object Model (DOM)

DOM ist eine Entwicklung des W3C und wird von vielen Programmiersprachen unterstützt. Das Standard-DOM ist so konzipiert, dass es unabhängig von einer Programmiersprache ist und eine strikte Hierarchie erzeugt. DOM definiert eine Reihe von Schnittstellen, die durch konkrete Programmiersprachen implementiert werden.

8.3.2 Simple API for XML Parsing (SAX)

SAX ist zum schnellen Verarbeiten der Daten von David Megginson als Public Domain entworfen worden. SAX ist im Gegensatz zu DOM nicht so speicherhungrig, weil das XML-Dokument nicht vollständig im Speicher abgelegt ist, und daher auch für sehr große Dokumente geeignet. Da SAX auf einem Ereignismodell basiert, wird die XML-Datei wie ein Datenstrom gelesen, und für erkannte Elemente wird ein Ereignis ausgelöst. Dies ist aber mit dem Nachteil verbunden, dass wahlfreier Zugriff auf ein einzelnes Element nicht ohne Zwischenspeicherung möglich ist.

8.3.3 Pull-API StAX

Im Gegensatz zu SAX, bei dem Methoden bereitgestellt werden, die beim Parsen aufgerufen werden, wird bei der Pull-API wie StAX aktiv der nächste Teil eines XML-Dokuments angefordert. Das Prinzip entspricht dem Iterator-Design-Pattern, das auch von der Collection-API bekannt ist. Es werden die beiden grundsätzlichen Verarbeitungsmodelle *Iterator* und *Cursor* unterschieden. Die Verarbeitung mit dem Iterator ist flexibler, aber auch ein bisschen aufwändiger. Die Cursor-Verarbeitung ist einfacher und schneller, aber nicht so flexibel. Beide Formen sind sich sehr ähnlich. Später werden beide Verfahren vorgestellt.

8.3.4 Java Document Object Model (JDOM)

JDOM ist eine einfache Möglichkeit, XML-Dokumente leicht und effizient mit einer schönen Java-API zu nutzen. Die aktuelle Entwicklung von JDOM geht von Jason Hunter und Rolf Lear aus, die erste Version hat Brett McLaughlin mitgestaltet.

Im Gegensatz zu SAX und DOM, die unabhängig von einer Programmiersprache sind, wurde JDOM speziell für Java entwickelt. Während das Original-DOM keine Rücksicht auf die Java-Datenstrukturen nimmt, nutzt JDOM konsequent die Collection-API. Auch ermöglicht JDOM eine etwas bessere Performance und eine bessere Speichernutzung als das Original-DOM.

Warum behandele ich JDOM in diesem Buch?

Das Original-W3C-DOM für Java ist historisch am ältesten, und JDOM war eine der ersten alternativen Java-XML-APIs. Mittlerweile steht JDOM nicht mehr alleine als W3C-DOM-Alternative da, und APIs wie *dom4j* (*http://dom4j.sourceforge.net/*) oder *XOM* (*http://www.xom.nu/*) gesellen sich dazu. Obwohl es um die Entwicklung von JDOM lange Zeit still war, zählt JDOM immer noch zu den populärsten[3] XML-APIs, wohl auch wegen der üppigen Dokumentation.

8.3.5 JAXP als Java-Schnittstelle zu XML

Die angesprochenen Technologien wie DOM, SAX, XPath, StAX sind erst einmal pure APIs. Für die APIs sind grundsätzlich verschiedene Implementierungen denkbar, jeweils mit Schwerpunkten wie Performance, Speicherverbrauch, Unicode-4-Unterstützung usw. Zwei Parser-Implementierungen sind zum Beispiel:

- **Xerces** (*http://tutego.de/go/xerces*): Die Standardimplementierung ab JDK 5. XSL-Stylesheet-Transformationen werden standardmäßig über einen *Compiling XSLT Processor* (XSLTC) verarbeitet.
- **Crimson** (*http://tutego.de/go/crimson*): Die Referenzimplementierung in Java 1.4.

Java API for XML Parsing (JAXP)

Der Nachteil bei der direkten Nutzung der Parser ist die Abhängigkeit von bestimmten Klassen. Daher wurde eine API mit dem Namen *Java API for XML Parsing* (JAXP) entworfen, die als Abstraktionsschicht über folgenden Technologien liegt:

- XML 1.0, XML 1.1
- DOM Level 3
- W3C XML Schema 1.0
- XSLT 1.0
- XInclude 1.0
- XPath 1.0
- SAX 2.0.2
- StAX 1.2 (JSR-173)

Die Parser validieren mit DTD oder einem W3C-XML-Schema und können mit *XInclude* Dokumente integrieren. Von DOM werden *DOM Level 3 Core* und *DOM Level 3 Load and Save* unterstützt.

JAXP ist schon lange ein Bestandteil der Java SE. Java 7 aktualisiert auf JAXP 1.4.5 und im Java SE 7 Update 40 auf JAXP 1.5.0, was auch Teil von Java 8 ist. Mehr Informationen zu den Versionen und Implementierungen gibt die Webseite *http://jaxp.java.net/*.

3 Laut *http://www.servlets.com/polls/results.tea?name=doms*.

Mit JAXP können Entwickler also einfach zwischen verschiedenen Parsern und XSLT-Transformern wählen, ohne den eigentlichen Code zu verändern. Das ist das gleiche Prinzip wie bei den Datenbanktreibern.

8.3.6 DOM-Bäume einlesen mit JAXP *

Um einen DOM-Baum einzulesen, soll unser folgendes Beispiel mit JAXP arbeiten. Eine Fabrik liefert uns einen XML-Parser, sodass wir den DOM-Baum einlesen können:

Listing 8.5 com/tutego/insel/xml/dom/DOMParty.java, main()

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
try ( InputStream in = Files.newInputStream( Paths.get( "party.xml" ) ) ) {
  Document document = builder.parse( in );
  System.out.println( document.getFirstChild().getTextContent() );
}
```

Die Parser sind selbstständig bei `DocumentBuilderFactory` angemeldet, und `newInstance()` liefert eine Unterklasse des `DocumentBuilder`.

8.4 Java Architecture for XML Binding (JAXB)

Java Architecture for XML Binding (JAXB) ist eine API zum Übertragen von Objektzuständen auf XML-Dokumente und umgekehrt. Anders als eine manuelle Abbildung von Java-Objekten auf XML-Dokumente oder das Parsen von XML-Strukturen und Übertragen der XML-Elemente auf Geschäftsobjekte arbeitet JAXB automatisch. Die Übertragungsregeln definieren Annotationen, die Entwickler selbst an die JavaBeans setzen können, aber JavaBeans werden gleich zusammen mit den Annotationen von einem Werkzeug aus einer XML-Schema-Datei generiert.

JAXB läuft in anderen Versionsnummern als das JDK. Java 7 aktualisiert auf JAXB 2.2, die gleiche Version, die auch Java 8 hat.

8.4.1 Bean für JAXB aufbauen

Wir wollen einen `Player` deklarieren, und JAXB soll ihn anschließend in ein XML-Dokument übertragen:

Listing 8.6 com/tutego/insel/xml/jaxb/Player.java, Player

```
@XmlRootElement
class Player {

  private String name;
```



```
  private Date   birthday;

  public String getName() {
    return name;
  }

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

  public void setBirthday( Date birthday ) {
    this.birthday = birthday;
  }

  public Date getBirthday() {
    return birthday;
  }
}
```

Die Klassen-Annotation `@XmlRootElement` ist an der JavaBean nötig, wenn die Klasse das Wurzelelement eines XML-Baums bildet. Die Annotation stammt aus dem Paket `javax.xml.bind.annotation`.

8.4.2 Utility-Klasse JAXB

Ein kleines Testprogramm baut eine Person auf und bildet sie dann in XML ab – die Ausgabe der Abbildung kommt auf den Bildschirm. Um ein Objekt in seine XML-Repräsentation zu bringen, lässt sich auf eine einfache statische `marshall(…)`-Methode der Utility-Klasse `JAXB` zurückgreifen:

Listing 8.7 com/tutego/insel/xml/xml/jaxb/SimplePlayerMarshaller.java, main()

```
Player johnPeel = new Player();
johnPeel.setName( "John Peel" );
johnPeel.setBirthday( new GregorianCalendar(1939,Calendar.AUGUST,30).getTime() );
JAXB.marshal( johnPeel, System.out );
```

JAXB beachtet standardmäßig alle Bean-Properties, also `birthday` und `name`, und nennt die XML-Elemente nach den Properties:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<player>
    <birthday>1939-08-30T00:00:00+01:00</birthday>
    <name>John Peel</name>
</player>
```

Die `marshall(…)`-Methode schreibt das in XML transformierte `Player`-Objekt in unserem Beispiel in den Konsolenausgabestrom. Die `JAXB`-Klasse bietet daneben auch `unmarshall(…)`-Methoden, um das Objekt wiederherzustellen, das schauen wir uns im nächsten Beispiel an.

Falls das Schreiben oder Lesen misslingt, gibt es eine `RuntimeException`. Diese referenziert die intern geprüften Ausnahmen vom Typ `DataBindingException`.

`class javax.xml.bind.`**`JAXB`**

- `static void marshal(Object jaxbObject, File xml)`
- `static void marshal(Object jaxbObject, OutputStream xml)`
- `static void marshal(Object jaxbObject, Result xml)`
- `static void marshal(Object jaxbObject, String xml)`
- `static void marshal(Object jaxbObject, URI xml)`
- `static void marshal(Object jaxbObject, URL xml)`
- `static void marshal(Object jaxbObject, Writer xml)`
  Schreibt das XML-Dokument in die angegebene Ausgabe. Im Fall von `URI`/`URL` wird ein HTTP-POST gestartet. Ist der Parametertyp `String`, wird er als URL gesehen und führt ebenfalls zu einem HTTP-Zugriff. `Result` ist ein Typ für eine XML-Transformation und wird später vorgestellt.
- `static <T> T unmarshal(File xml, Class<T> type)`
- `static <T> T unmarshal(InputStream xml, Class<T> type)`
- `static <T> T unmarshal(Reader xml, Class<T> type)`
- `static <T> T unmarshal(Source xml, Class<T> type)`
- `static <T> T unmarshal(String xml, Class<T> type)`
- `static <T> T unmarshal(URI xml, Class<T> type)`
- `static <T> T unmarshal(URL xml, Class<T> type)`
  Rekonstruiert aus der gegebenen XML-Quelle den Java-Objektgraphen.

8.4.3 Ganze Objektgraphen schreiben und lesen

JAXB bildet nicht nur das zu schreibende Objekt ab, sondern auch rekursiv alle referenzierten Unterobjekte. Wir wollen den Spieler dazu in einen Raum setzen und den Raum in XML abbilden. Dazu muss der Raum die Annotation `@XmlRootElement` bekommen, und bei `Player` kann sie entfernt werden, wenn nur der Raum selbst, aber keine `Player` als Wurzelobjekte zum Marshaller kommen:

Listing 8.8 com/tutego/insel/xml/xml/jaxb/Room.java, Room

```
@XmlRootElement( namespace = "http://tutego.com/" )
public class Room {

  private List<Player> players = new ArrayList<>();

  @XmlElement( name = "player" )
  public List<Player> getPlayers() {
    return players;
  }

  public void setPlayers( List<Player> players ) {
    this.players = players;
  }
}
```

Zwei Annotationen kommen vor: Da `Room` der Start des Objektgraphen ist, trägt es `@XmlRootElement`. Als Erweiterung ist das Element `namespace` für den Namensraum gesetzt, da bei eigenen XML-Dokumenten immer ein Namensraum genutzt werden soll. Weiterhin ist eine Annotation `@XmlElement` am Getter `getPlayers()` platziert, um den Namen des XML-Elements zu überschreiben, damit das XML-Element nicht `<players>` heißt, sondern `<player>`.

Kommen wir abschließend zu einem Beispiel, das einen Raum mit zwei Spielern aufbaut und diesen Raum dann in eine XML-Datei schreibt:

Listing 8.9 com/tutego/insel/xml/jaxb/RoomMarshaller.java, main()

```
Player john = new Player();
john.setName( "John Peel" );

Player tweet = new Player();
tweet.setName( "Zwitscher Zoe" );

Room room = new Room();
room.setPlayers( Arrays.asList( john, tweet ) );

Path path = Paths.get( "room.xml" );
try ( Writer out = Files.newBufferedWriter( path, StandardCharsets.UTF_8 ) ) {
  JAXB.marshal( room, out );

  Room room2 = JAXB.unmarshal( path.toFile(), Room.class );
  System.out.println( room2.getPlayers().get( 0 ).getName() ); // John Peel
}
```

```
catch ( IOException e ) {
  e.printStackTrace();
}
```

Die Ausgabe ist:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:room xmlns:ns2="http://tutego.com/">
    <player>
        <name>John Peel</name>
    </player>
    <player>
        <name>Zwitscher Zoe</name>
    </player>
</ns2:room>
```

Da beim Spieler das Geburtsdatum nicht gesetzt war (`null` wird referenziert), wird es auch nicht in XML abgebildet.

8.4.4 JAXBContext und Marshaller/Unmarshaller nutzen

Die Utility-Klasse `JAXB` verfügt ausschließlich über statische überladene `marshal(…)`- und `unmarshal(…)`-Methoden, ist aber in Wirklichkeit nur eine Fassade. Vielmehr ist `JAXBContext` das tatsächliche Ausgangsobjekt, und davon erfragt werden ein `Marshaller` zum Schreiben und ein `Unmarshaller` zum Lesen:

Listing 8.10 com/tutego/insel/xml/xml/jaxb/PlayerMarshaller.java, main()

```
Player johnPeel = new Player();
johnPeel.setName( "John Peel" );
johnPeel.setBirthday( new GregorianCalendar(1939,Calendar.AUGUST,30).getTime() );
JAXBContext context = JAXBContext.newInstance( Player.class );
Marshaller m = context.createMarshaller();
m.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE );
m.marshal( johnPeel, System.out );
```

Die Ausgabe ist identisch mit der ersten:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<player>
    <birthday>1939-08-30T00:00:00+01:00</birthday>
    <name>John Peel</name>
</player>
```

Alles bei JAXB beginnt mit der zentralen Klasse `JAXBContext`. Die statische Methode `JAXBContext.newInstance(Class...)` erwartet standardmäßig eine Aufzählung der Klassen, die JAXB behandeln soll; die Klassennamen können auch voll qualifiziert über einen String angegeben werden mit `newInstance(String)`. Der `JAXBContext` liefert den `Marshaller` zum Schreiben und den `Unmarshaller` zum Lesen. Die Fabrikmethode `createMarshaller()` gibt dabei einen Schreiberling, der mit `marshal(Object, …)` das Wurzelobjekt in einen Datenstrom schreibt. Das zweite Argument von `marshal(Object, …)` ist unter anderem ein `OutputStream` (wie `System.out` in unserem Beispiel), `Writer` oder `File`-Objekt.

class javax.xml.bind.JAXBContext

- `static JAXBContext newInstance(Class... classesToBeBound) throws JAXBException`
  Liefert ein Exemplar vom `JAXBContext` mit Klassen, die als Wurzelklassen für JAXB verwendet werden können.
- `abstract Marshaller createMarshaller()`
  Erzeugt einen `Marshaller`, der Java-Objekte in XML-Dokumente konvertieren kann.
- `abstract Unmarshaller createUnmarshaller()`
  Erzeugt einen `Unmarshaller`, der XML-Dokumente in Java-Objekte konvertiert.

class javax.xml.bind.Marshaller

- `void marshal(Object jaxbElement, File output)`
- `void marshal(Object jaxbElement, OutputStream os)`
- `void marshal(Object jaxbElement, Writer writer)`
  Schreibt den Objektgraphen von `jaxbElement` in eine Datei oder einen Ausgabestrom.
- `void marshal(Object jaxbElement, Node node)`
  Erzeugt vom Objekt einen DOM-Knoten. Der kann dann in ein XML-Dokument gesetzt werden.
- `void marshal(Object jaxbElement, XMLEventWriter writer)`
- `void marshal(Object jaxbElement, XMLStreamWriter writer)`
  Erzeugt für ein `jaxbElement` einen Informationsstrom für den `XMLEventWriter` bzw. `XMLStreamWriter`. Die StAX-Klassen werden später genauer vorgestellt.
- `void setProperty(String name, Object value)`
  Setzt eine Eigenschaft der `Marshaller`-Implementierung. Eine Einrückung etwa setzt das Paar `Marshaller.JAXB_FORMATTED_OUTPUT`, `Boolean.TRUE`.

Der `Unmarshaller` bietet über zehn Varianten von `unmarshal(…)`, unter anderem mit den Parametertypen `File`, `InputSource`, `InputStream`, `Node`, `Reader`, `URL`, `XMLEventReader`.

Tipps uns Tricks mit JAXBContext

Den `JAXBContext` aufzubauen kostet Zeit und Speicher. Er sollte daher für wiederholte Operationen gespeichert werden. Noch eine Information: `Marshaller` und `Unmarshaller` sind nicht threadsicher; es darf keine zwei Threads geben, die gleichzeitig den `Marshaller`/`Unmarshaller` nutzen.

Die `unmarshal(…)`-Methode ist überladen mit Parametern, die typische Datenquellen repräsentieren, etwa Dateien oder Eingabeströme wie ein `Reader`. Allerdings sind noch andere Parametertypen interessant, und es lohnt sich, hier einmal in die API-Dokumentation zu schauen. Ein spannender Typ ist `javax.xml.transform.Source` bzw. die Implementierung der Schnittstelle durch `JAXBSource`. `JAXBSource` ist die Quelle, aus der JAXB seine Informationen bezieht, um ein neues Java-Objekt zu rekonstruieren.

Das nächste Beispiel nimmt sich das aus dem vorangehenden Beispiel aufgebaute Objekt `room` als Basis und erzeugt eine tiefe Kopie davon:

Listing 8.11 com/tutego/insel/xml/xml/jaxb/RoomCopy.java, main() Ausschnitt

```
Room room = …
JAXBContext context = JAXBContext.newInstance( Room.class );
Unmarshaller unmarshaller = context.createUnmarshaller();
JAXBSource source = new JAXBSource( context, room );
Room copiedRoom = Room.class.cast( unmarshaller.unmarshal( source ) );
System.out.println( copiedRoom.getPlayers() ); // [com.tutego.insel.xml.jaxb.Player@...]
```

Das Beispiel zeigt somit, wie sich mithilfe von JAXB Objektkopien erzeugen lassen.

8.4.5 Validierung

Beim Schreiben und auch beim Lesen von XML-Dokumenten kann JAXB diese gegen ein XML-Schema validieren:

- Falsche XML-Dokumente sollen nicht eingelesen werden. Wenn die Schema-Datei zum Beispiel vorschreibt, dass eine Zahl (etwa für die Bohrtiefe) nur im Wertebereich von 0 bis 10.000 liegen darf, und in einer XML-Datei taucht dann ein Wert von 10.600 auf, dann wäre die XML-Datei nach diesem Schema nicht valide. JAXB sollte sie ablehnen.
- Falsche Werte in JavaBeans dürfen nicht zu nichtvaliden XML-Dokumenten führen. JavaBeans bestehen aus einfachen Settern und Gettern, und die Prüfungen im XML-Schema gehen weit über das hinaus, was üblicherweise eine JavaBean prüft. So kann eine Schema-Definition etwa vorschreiben, dass eine Zeichenkette nach einem ganz speziellen regulären Ausdruck geschrieben wird. In der Regel sind die Setter aber nicht so implementiert, dass sie die Strings direkt prüfen. JAXB sollte es auch nicht erlauben, dass JavaBeans mit falschen Strings geschrieben werden und zu nichtvaliden XML-Dokumenten führen können.

Zentral bei der Validierung ist eine XML-Schema-Datei. Doch wo kommt diese her? Wurden aus der Schema-Datei die JavaBeans generiert, ist logischerweise die Schema-Beschreibung schon da. Sind die JavaBeans aber zuerst da, gibt es erst einmal keine Schema-Datei. Über unterschiedliche Wege lässt sich eine passende Schema-Datei entwickeln:

- **Per Hand**: Die XML-Schema-Datei wird per Hand selbstständig ohne Tool entwickelt.
- **Über die XML-Dokumente**: Ein Tool analysiert XML-Dateien und erzeugt aufgrund der erkannten Strukturen eine XML-Schema-Datei. Für dieses Verfahren gibt es eine Reihe von Tools, und einige sind auch online verfügbar, etwa *http://tutego.de/go/xml-2-xsd*.
- **Mit schemagen**: Das JDK bringt ein Tool mit dem Namen *schemagen* mit, das auf JAXB-annotierte Beans angesetzt wird und die Schema-Datei generiert.

Mit schemagen aus JAXB-Beans eine XML-Schema-Datei generieren

Das Tool *schemagen* befindet sich wie alle anderen JDK-Tools im *bin*-Verzeichnis. Das Kommandozeilentool erwartet die Angabe einer Quellcodeklasse oder einer compilierten Klasse und spuckt die Schema-Beschreibungen aus. Kapseln wir das in ein Skript:

Listing 8.12 room-schemagen.bat

```
@echo off
PATH=%PATH%;C:\Program Files\Java\jdk1.8.0\bin
schemagen com.tutego.insel.xml.jaxb.Room
```

Und rufen wir es auf:

```
$ room-schemagen.bat
Note: Writing C:\...\schema1.xsd
Note: Writing C:\...\schema2.xsd
```

Das Tool erzeugt zwei Schema-Dateien, und sie sehen so aus:

Listing 8.13 schema1.xsd

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" targetNamespace="http://tutego.com/"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:import schemaLocation="schema2.xsd" />

  <xs:element name="room" type="room" />

</xs:schema>
```

Listing 8.14 schema2.xsd

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="player" type="player" />
```

```
  <xs:complexType name="room">
    <xs:sequence>
      <xs:element ref="player" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="player">
    <xs:sequence>
      <xs:element name="birthday" type="xs:dateTime" minOccurs="0" />
      <xs:element name="name" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Ein genaues Verständnis des Schemas ist nicht nötig.

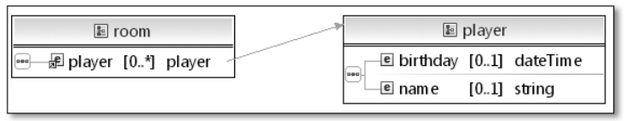

Abbildung 8.1 Visualisierung des Schemas

Schema-Validierung mit JAXB

Wir wollen die Validierungsmöglichkeiten von JAXB für unsere bekannte Raum-Datei ausprobieren und bewusst zwei Fehler einbauen (siehe Tabelle 8.4).

Valide XML-Datei (nach Schema)	Nichtvalide XML-Datei (nach Schema)
Listing 8.15 room.xml	**Listing 8.16** room-invalid.xml

`<?xml version="1.0" encoding="UTF-8"?>`
`<ns2:room xmlns:ns2="http://tutego.com/">`
`  <player>`
`  <name>John Peel</name>`
`  </player>`
`  <player>`
`  <name>Zwitscher Zoe</name>`
`  </player>`
`</ns2:room>`
	`<?xml version="1.0" encoding="UTF-8"?>`
	`<room>`
	`  <player>`
	`  <name>John Peel</name>`
	`  </player>`
	`  <player>`
	`  <name>Zwitscher Zoe</name>`
	`  <name>Heini Hayward</name>`
	`  </player>`
	`</room>`

Tabelle 8.4 Nach dem Schema valide und nichtvalide XML-Datei

Das Java-Programm aus dem vorigen Abschnitt schrieb eine korrekte XML-Datei *room.xml*. In *room-invalid.xml* fehlt einmal der Namensraum, und dann sind zwei Namen angegeben, obwohl die Schema-Datei nur einen Namen erlaubt.

Damit JAXB den Fehler erkennt, muss es mit der neuen Schema-Datei verbunden werden. JAXP hat eine eigene API für Validierungen, die dafür eingesetzt wird (mehr zur Schema-Validierung folgt später in Abschnitt 8.9, »XML-Schema-Validierung*«):

Listing 8.17 com/tutego/insel/xml/jaxb/ValidatingRoomUnmarshaller.java, main()

```
JAXBContext context = JAXBContext.newInstance( Room.class );
Unmarshaller unmarshaller = context.createUnmarshaller();
SchemaFactory sf = SchemaFactory.newInstance( W3C_XML_SCHEMA_NS_URI );
Schema schema = sf.newSchema(
  ValidatingRoomUnmarshaller.class.getResource( "/schema1.xsd" ) );
unmarshaller.setSchema( schema );
URL url = Paths.get( "invalid-room.xml" ).toUri().toURL();
Room room = (Room) unmarshaller.unmarshal( url );
System.out.println( room.getPlayers() );
```

Es wird ein Exemplar eines `Schema`-Objekts erzeugt und dieses über `setSchema(Schema)` beim `Unmarshaller` gesetzt. (Achtung: `JAXB.unmarshal(file, Room.class)` wird nicht funktionieren!)

Schon der erste Fehler in *invalid-room.xml* führt zum Abbruch:

```
Exception in thread "main" javax.xml.bind.UnmarshalException
 - with linked exception:
[org.xml.sax.SAXParseException; systemId: ➲
file:/C:/Insel/programme/18_XML/invalid-room.xml; lineNumber: 2; columnNumber: 7; ➲
cvc-elt.1: Cannot find the declaration of element 'room'.]
```

Ist der Fehler behoben, kommt das zweite Problem zum Tragen, nämlich der Umstand, dass es zwei Namen gibt:

```
Exception in thread "main" javax.xml.bind.UnmarshalException
 - with linked exception:
[org.xml.sax.SAXParseException; systemId: ➲
file:/C:/Insel/programme/18_XML/invalid-room.xml; lineNumber: 8; columnNumber: 11; ➲
cvc-complex-type.2.4.d: Invalid content was found starting with element 'name'. ➲
No child element is expected at this point.]
```

Erst wenn der Fehler behoben wurde, gibt es keine Ausnahme mehr, und JAXB gibt Ruhe.

8.4.6 Weitere JAXB-Annotationen *

XML-Schemas können recht komplex werden, sodass auch die Anzahl der JAXB-Annotationen und Möglichkeiten hoch ist. Im Folgenden sollen verschiedene JAXB-Annotationen ihre Wirkung auf die XML-Ausgaben zeigen.

Zugriff über Setter/Getter oder Attribute

JAXB kann sich die Werte über JavaBean-Properties – also Setter/Getter – setzen und lesen und/oder direkt auf die Attribute zugreifen. Der Attributzugriff ist vergleichbar mit der Standardserialisierung, und der Zugriff über die Property ist von der JavaBeans Persistence über `java.beans.XMLEncoder/java.beans.XMLDecoder` realisiert. Welchen Weg JAXB gehen soll, bestimmt die Annotation `@XmlAccessorType`, die üblicherweise an der Klasse festgemacht wird. Drei Werte sind interessant:

Wert	Datenaustausch über
`@XmlAccessorType(XmlAccessType.FIELD)`	jedes nichtstatische Attribut
`@XmlAccessorType(XmlAccessType.PROPERTY)`	jede JavaBean-Property
`@XmlAccessorType(XmlAccessType.PUBLIC_MEMBER)`	jede öffentliche JavaBean-Property oder jedes öffentliche Attribut

Tabelle 8.5 Womit JAXB die Daten austauscht

Die Standardbelegung ist `AccessType.PUBLIC_MEMBER`.

@Transient

Die Annotation `@Transient` nimmt ein Element von der XML-Abbildung aus. Das ist nützlich für den `XmlAccessType.FIELD` oder `XmlAccessType.PROPERTY`, da dann auch private Eigenschaften geschrieben werden, was allerdings nicht in jedem Fall erwünscht ist:

Java-Beispiel	XML-Ergebnis

`class Person {`
` **@XmlTransient** public int id;`
` public String firstname;`
` public String lastname;`
`}`
	`<person>`
	` <firstname>Christian</firstname>`
	` <lastname>Ullenboom</lastname>`
	`</person>`

Tabelle 8.6 Beispiel für @Transient und Ergebnis

Werte als Attribute schreiben mit @XmlAttribute

Üblicherweise schreibt JAXB jeden Wert in ein eigenes XML-Element. Soll der Wert als Attribut geschrieben werden, kommt die Annotation `@XmlAttribute` zum Einsatz:

Java-Beispiel	XML-Ergebnis

`class Person {`
` public String name;`
` public **@XmlAttribute** int id;`
`}`
	`<person id="123">`
	` <name>Christian</name>`
	`</person>`

Tabelle 8.7 Beispiel für @XmlAttribute und Ergebnis

Reihenfolge der Elemente ändern

Ist die Reihenfolge der XML-Elemente wichtig, so lässt sich mit dem `propOrder` von `@XmlType` die Reihenfolge der Eigenschaften bestimmen:

Java-Beispiel	XML-Ergebnis

`class Person {`
` public String lastname,`
`  firstname;`
`}`
	`<person>`
	` <lastname>Ullenboom</lastname>`
	` <firstname>Christian</firstname>`
	`</person>`

`**@XmlType(**`
` **propOrder = { "firstname", "lastname" }**`
`)`
`class Person {`
` public String lastname, firstname;`
`}`
	`<person>`
	` <firstname>Christian</firstname>`
	` <lastname>Ullenboom</lastname>`
	`</person>`

Tabelle 8.8 XML-Abbildung ohne und mit @XmlType(propOrder...)

Einzelner Wert ohne eigenes XML-Element

Gibt es nur ein Element in der Klasse, so kann `@XmlValue` es direkt ohne Unterelement in den Rumpf setzen:

Java-Beispiel	XML-Ergebnis

`class Person {`
` public int id;`
`}`
	`<person>`
	` <id>123</id>`
	`</person>`

`class Person {`
` public **@XmlValue** int id;`
`}`
	`<person>123</person>`

Tabelle 8.9 XML-Abbildung mit @XmlValue

Kompakte Listendarstellung

Die Datenstruktur Liste wird in JAXB üblicherweise so abgebildet, dass jedes Listenelement einzeln in ein XML-Element kommt. Die Annotation `@XmlList` weist JAXB an, Elemente einer Sammlung mit Leerzeichen zu trennen. Das funktioniert gut bei IDs, aber natürlich nicht mit allgemeinen Zeichenketten, die Leerzeichen enthalten:

Java-Beispiel	XML-Ergebnis

`class Person {`
`  public List<String> emails;`
`}`
	`<person>`
	`  <emails>muh@kuh.de</emails>`
	`  <emails>zick@zak.com</emails>`
	`</person>`

`class Person {`
`  public @XmlList List<String> emails;`
`}`
	`<person>`
	`  <emails>muh@kuh.de zick@zak.com</emails>`
	`</person>`

Tabelle 8.10 Beispiel für @XmlList

Elemente zusätzlich einpacken

Die Annotation `@XmlElementWrapper` dient dazu, ein zusätzliches XML-Element zu erzeugen. In der Regel wird das für Sammlungen angewendet, wie auch das folgende Beispiel zeigt:

Java-Beispiel	XML-Ergebnis

`class Person {`
`  public List<String> emails;`
`}`
	`<person>`
	`  <emails>muh@kuh.de</emails>`
	`  <emails>zick@zack.com</emails>`
	`</person>`

`class Person {`
`  @XmlElementWrapper(name = "emails")`
`  @XmlElement(name = "email")`
`  public List<String> emails;`
`}`
	`<person>`
	`  <emails>`
	`  <email>muh@kuh.de</email>`
	`  <email>zick@zack.com</email>`
	`  </emails>`
	`</person>`

Tabelle 8.11 Beispiel für @XmlElementWrapper/@XmlElement

Anpassen der XML-Abbildung

Nicht immer passt die Standardabbildung eines Datentyps gut. Für Farben sollen zum Beispiel nicht die Rot-, Grün- und Blau-Werte einzeln geschrieben werden, sondern alles kompakt in einem String. Auch die Standardabbildung für Datumswerte trifft nicht jeden Geschmack:

Java-Beispiel	XML-Ergebnis

`class Person {`
`  public Date birthday;`
`}`
	`<person>`
	`  <birthday>1973-03-12T00:00:00+01:00</birthday>`
	`</person>`

Tabelle 8.12 Standard-XML-Abbildung eines Datums

Für Aufgaben dieser Art erlaubt die Annotation `@XmlJavaTypeAdapter` die Angabe einer Konverterklasse, die einmal den Weg vom Objekt in eine String-Repräsentation für das XML-Element und dann vom String in das Objekt zurück beschreibt:

```
class Person {
  @XmlJavaTypeAdapter( DateAdapter.class )
  public Date birthday;
}
```

Die eigene Klasse `DateAdapter` erweitert die vorgegebene JAXB-Klasse `XmlAdapter` und überschreibt zwei Methoden für beide Konvertierungswege:

```
class DateAdapter extends XmlAdapter<String, Date> {
  private final static DateFormat formatter = new SimpleDateFormat( "dd/MM/yyyy" );

  public Date unmarshal( String date ) throws ParseException {
    return formatter.parse( date );
  }

  public String marshal( Date date ) {
    return formatter.format( date );
  }
}
```

Damit bekommt die Ausgabe das gewünschte Format:

```
<person>
    <birthday>12/03/1973</birthday>
</person>
```

```
abstract class javax.xml.bind.annotation.adapters.XmlAdapter<ValueType,BoundType>
```

- `abstract ValueType marshal(BoundType v)`
  Konvertiert `v` in einen Werttyp, der dann in eine XML-Repräsentation überführt wird.
- `abstract BoundType unmarshal(ValueType v)`
  Überführt den Wert in den XML-Typ.

ValueType und BoundType sind Typvariablen, aber ungewöhnlicherweise sind es keine einfachen Großbuchstaben.

Der spezielle Datentyp XMLGregorianCalendar

Neben der Möglichkeit, Datumswerte mit einem @XmlJavaTypeAdapter/XmlAdapter zu übersetzen, bietet JAXB den speziellen abstrakten Datentyp XMLGregorianCalendar. Die Abbildung in XML ist kompakter:

Java-Beispiel	XML-Ergebnis

class Person {
 public **XMLGregorianCalendar** birthday;
}
	<person>
	 <birthday>1973-03-12</birthday>
	</person>

Tabelle 8.13 Standardabbildung von XMLGregorianCalendar

XMLGregorianCalendar wird auch automatisch von dem Werkzeug xjc genutzt, wenn in der XML-Schema-Datei ein Datum vorkommt. Nicht ganz einfach sind die Erzeugung eines XMLGregorianCalendar-Objekts und die Belegung – hier gibt es noch Potenzial für Verbesserungen:

```
Person p = new Person();
GregorianCalendar c = new GregorianCalendar( 1973, Calendar.MARCH, 12 );
XMLGregorianCalendar gc = DatatypeFactory.newInstance().newXMLGregorianCalendar( c );
gc.setTimezone( DatatypeConstants.FIELD_UNDEFINED );
gc.setTime( DatatypeConstants.FIELD_UNDEFINED,
            DatatypeConstants.FIELD_UNDEFINED,
            DatatypeConstants.FIELD_UNDEFINED );
p.birthday = gc;
```

Die abstrakte Klasse DatatypeFactory bietet weitere statische Methoden für Mapper-Objekte, die XML in Objekte überführen oder umgekehrt:

```
abstract class javax.xml.datatype.DatatypeFactory
```

- static DatatypeFactory newInstance()
  Liefert eine DatatypeFactory-Implementierung.
- abstract XMLGregorianCalendar newXMLGregorianCalendar()
  Liefert einen XMLGregorianCalendar, bei dem alle Werte undefiniert sind.
- abstract XMLGregorianCalendar newXMLGregorianCalendar(GregorianCalendar cal)
- XMLGregorianCalendar newXMLGregorianCalendarDate(int year, int month, int day, int timezone)
- XMLGregorianCalendar newXMLGregorianCalendarTime(int hours, int minutes, int seconds, int timezone)

- XMLGregorianCalendar newXMLGregorianCalendarTime(int hours, int minutes, int seconds, BigDecimal fractionalSecond, int timezone)
- XMLGregorianCalendar newXMLGregorianCalendar(int year, int month, int day, int hour, int minute, int second, int millisecond, int timezone)
- abstract XMLGregorianCalendar newXMLGregorianCalendar(BigInteger year, int month, int day, int hour, int minute, int second, BigDecimal fractionalSecond, int timezone)
  Liefert ein XMLGregorianCalendar-Objekt mit unterschiedlichen Vorbelegungen.

Weiterhin gibt es newDuration(…)-Methoden, die javax.xml.datatype.Duration-Objekte liefern. Die Duration-Objekte können auf XMLGregorianCalendar aufaddiert werden bzw. repräsentieren in XML-Schema-Dateien den Typ xs:duration.

Beispiel

Eine XML-Schema-Datei soll für das Element period eine Dauer definieren:

```
<xs:element name="period" type="xs:duration"/>
```

Angewendet kann es so aussehen, um die Dauer von einem Jahr und einem Monat anzugeben:

```
<period>P1Y1M</period>
```

Werden aus Schema-Dateien die JavaBeans automatisch generiert, wird Duration für xs:duration eingesetzt.

Hierarchien einsetzen

Die XML-Abbildung von Objekten, die in Klassenbeziehungen organisiert sind, bedarf einer besonderen Vorbereitung. Nehmen wir an, Player und Key sind zwei Klassen, die von GameObject abgeleitet sind (eine Schnittstelle wäre für JAXB auch möglich). Unser Ziel ist es, Spieler und Schlüssel in einen Raum zu setzen:

```
abstract class GameObject {
  public String name;
}

@XmlRootElement public class Player extends GameObject { }

@XmlRootElement public class Key extends GameObject {
  public int id;
}
```

Zunächst gilt, dass die konkreten Klassen die Annotation @XmlRootElement tragen müssen. Ein Beispielraum soll einen Spieler und einen Schlüssel beherbergen:

```
Player player= new Player();
player.name = "Chris";

Key key = new Key();
key.name = "Entretenimiento";
key.id = 12;

Room room = new Room();
room.objects.add( key );
room.objects.add( player );
```

Der Raum referenziert in einer Liste allgemeine Objekte vom Typ `GameObject`. Nun reicht im `Room` ein einfaches

```
public List<GameObject> objects = new ArrayList<GameObject>();
```

zum Halten der Objektverweise aber nicht aus! Beim Verarbeiten würde JAXB die Information fehlen, welches Element denn tatsächlich in der Liste ist, denn ein `Player` sollte ja etwa durch `<player>` beschrieben sein und ein Schlüssel durch `<key>`. Die Abbildung kann nicht `<objects>` lauten, denn beim Lesen muss ein konkreter Untertyp rekonstruiert werden; wenn JAXB beim Lesen ein `<objects>` sieht, weiß es erst einmal nicht, ob ein `Player` oder ein `Key` zu erzeugen und in die Liste zu hängen ist. Das Ziel ist aber die folgende Abbildung:

```
<room>
  <key>
    <name>Entretenimiento</name>
    <id>12</id>
  </key>
  <player>
    <name>Chris</name>
  </player>
</room>
```

Die Lösung liegt in der Anwendung der Annotationen `@XmlElementRefs` und `@XmlElementRef`. Ersteres ist ein Container, und das Zweite bestimmt den Typ, der in der Liste zu erwarten ist:

```
@XmlRootElement public class Room {
  @XmlElementRefs( {
    @XmlElementRef( type = Player.class ),
    @XmlElementRef( type = Key.class ),
  } )
  public List<GameObject> objects = new ArrayList<GameObject>();
}
```

Mit diesem Hinweis berücksichtigt JAXB den Typ der Kinder und schreibt nicht einfach `<objects>`. Die Elementtypen in der Sammlung sind von uns mit `@XmlRootElement` annotiert und

geben den Namen der XML-Elemente »player« und »key« vor (wir hätten natürlich mit so etwas wie `@XmlRootElement(name="sportsman")` den XML-Elementnamen überschreiben können).

8.4.7 Beans aus XML-Schema-Datei generieren

Da es für existierende XML-Dateien mühselig ist, die annotierten JavaBeans von Hand aufzubauen, gibt es einen Generator, genannt *Java Architecture for XML Binding Compiler*, kurz `xjc`.Er kann von der Kommandozeile, vom Ant-Skript oder auch von Entwicklungsumgebungen aus aufgerufen werden. Er nimmt eine XML-Schema-Datei und generiert die Java-Klassen und eine `ObjectFactory`, die als – wie der Name schon sagt – Fabrik für die gemappten Objekte aus den XML-Elementen fungiert.

xjc aufrufen

Im ersten Schritt wechseln wir auf die Kommandozeile und testen entweder, ob das *bin*-Verzeichnis vom JDK im Suchpfad ist, oder wir wechseln in das *bin*-Verzeichnis, sodass wir `xjc` direkt aufrufen können, und folgende Ausgabe erscheint:

```
$ xjc

Usage: xjc [-options ...] <schema file/URL/dir/jar> ... [-b <bindinfo>] ...
If dir is specified, all schema files in it will be compiled.
If jar is specified, /META-INF/sun-jaxb.episode binding file will be compiled.
Options:
  -nv                :  do not perform strict validation of the input schema(s)
  -extension         :  allow vendor extensions - do not strictly follow the
                        Compatibility Rules and App E.2 from the JAXB Spec
  -b <file/dir>      :  specify external bindings files (each <file> must have its own -b)
                        If a directory is given, **/*.xjb is searched
  -d <dir>           :  generated files will go into this directory
  -p <pkg>           :  specifies the target package
  -httpproxy <proxy> :  set HTTP/HTTPS proxy. Format is [user[:password]@]
                        proxyHost:proxyPort
  -httpproxyfile <f> :  Works like -httpproxy but takes the argument in a file to
                        protect password
  -classpath <arg>   :  specify where to find user class files
  -catalog <file>    :  specify catalog files to resolve external entity references
                        support TR9401, XCatalog, and OASIS XML Catalog format.
  -readOnly          :  generated files will be in read-only mode
  -npa               :  suppress generation of package level annotations
                        (**/package-info.java)
  -no-header         :  suppress generation of a file header with timestamp
  -target (2.0|2.1)  :  behave like XJC 2.0 or 2.1 and generate code that doesnt use
                        any 2.2 features.
```

```
  -enableIntrospection :  enable correct generation of Boolean getters/setters to
                          enable Bean Introspection apis
  -contentForWildcard  :  generates content property for types with multiple
                          xs:any derived elements
  -xmlschema           :  treat input as W3C XML Schema (default)
  -relaxng             :  treat input as RELAX NG (experimental,unsupported)
  -relaxng-compact     :  treat input as RELAX NG compact syntax (experimental,unsupported)
  -dtd                 :  treat input as XML DTD (experimental,unsupported)
  -wsdl                :  treat input as WSDL and compile schemas inside it
                          (experimental,unsupported)
  -verbose             :  be extra verbose
  -quiet               :  suppress compiler output
  -help                :  display this help message
  -version             :  display version information
  -fullversion         :  display full version information

Extensions:
  -Xinject-code        :  inject specified Java code fragments into the generated code
  -Xlocator            :  enable source location support for generated code
  -Xsync-methods       :  generate accessor methods with the 'synchronized' keyword
  -mark-generated      :  mark the generated code as @javax.annotation.Generated
  -episode <FILE>      :  generate the episode file for separate compilation
```

Eigentlich ist bis auf die Angabe der Schema-Quelle (aus einer Datei oder alternativ die URL) keine weitere Angabe nötig. Es ist aber praktisch, zwei Optionen zu setzen: `-p` bestimmt das Java-Paket für die generierten Klassen und `-d` das Ausgabeverzeichnis, in dem der Generator die erzeugten Dateien ablegt.

Wetterdaten von Open Weather Map beziehen

Für ein Beispiel wählen wir einen Dienst der Open Weather Map, eines frei zugänglichen Servers für Wetterdaten. Die Webseite *http://openweathermap.org/API* zeigt den Einsatz der einfachen API, bei der ganz simpel in der URL ein Ort kodiert ist, von dem die Wetterdaten erfragt werden sollen, etwa so: *http://api.openweathermap.org/data/2.5/find?q=Frankfurt&units=metric&mode=xml*. Alternativ lassen sich Grafiken aufbauen, nach Geo-Koordinaten suchen oder statt XML auch JSON produzieren – für unser Beispiel ist XML ganz gut, das für die Anfrage wie folgt aussieht:

```
<?xml version="1.0" encoding="utf-8"?>
<cities>
  <calctime>0.0027</calctime>
  <count>1</count>
  <mode>name</mode>
  <list>
    <item>
```



```
      <city id="2925533" name="Frankfurt am Main">
        <coord lon="8.68341" lat="50.1121"/>
        <country>Germany</country>
        <sun rise="2014-03-13T05:42:01" set="2014-03-13T17:27:13"/>
      </city>
      <temperature value="15.25" min="13.2" max="17" unit="celsius"/>
      <humidity value="38" unit="%"/>
      <pressure value="1026" unit="hPa"/>
      <wind>
        <speed value="0.5" name="Calm"/>
        <direction value="60" code="ENE" name="East-northeast"/>
      </wind>
      <clouds value="0" name="sky is clear"/>
      <precipitation mode="no"/>
      <weather number="800" value="Sky is Clear" icon="01n"/>
      <lastupdate value="2014-03-13T17:56:36" unix="1394733396"/>
    </item>
  </list>
</cities>
```

Für unser Beispiel wollen wir das XML-Dokument nicht von Hand auseinanderpflücken, sondern JAXB soll uns eine gefüllte JavaBean mit allen Informationen liefern. Leider gibt es für diese einfache XML-Datei kein XML-Schema, sodass wir uns mit einem Trick behelfen, nämlich aus einer XML-Datei ein Schema generieren lassen und dann mit diesem zu `xjc` gehen. Es gibt im Internet einige freie XML-nach-Schema-Konverter, für mein Beispiel nutze ich *http://www.xmlforasp.net/CodeBank/System_Xml_Schema/BuildSchema/BuildXMLSchema.aspx*. Damit die Datentypen korrekt erkannt werden, ändere ich in dem Beispiel die Windrichtung vom Ganzzahltyp `<direction value="60"…>` auf eine Fließkommazahl, etwa `value="60.123"`, sonst steht im generierten Schema der Typ `xsd:int`, was später zu Problemen führt, wenn der Dienst eine Fließkommazahl liefert.

```
<?xml version="1.0" encoding="utf-16"?>
<xsd:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="cities">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="calctime" type="xsd:decimal" />
        <xsd:element name="count" type="xsd:int" />
        <xsd:element name="mode" type="xsd:string" />
        <xsd:element name="list">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="item">
```

```
<xsd:complexType>
  <xsd:sequence>
    <xsd:element name="city">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="coord">
            <xsd:complexType>
              <xsd:attribute name="lon" type="xsd:decimal" />
              <xsd:attribute name="lat" type="xsd:decimal" />
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="country" type="xsd:string" />
          <xsd:element name="sun">
            <xsd:complexType>
              <xsd:attribute name="rise" type="xsd:dateTime" />
              <xsd:attribute name="set" type="xsd:dateTime" />
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:int" />
        <xsd:attribute name="name" type="xsd:string" />
      </xsd:complexType>
    </xsd:element>
  </xsd:complexType>
</xsd:element>
<xsd:element name="temperature">
  <xsd:complexType>
    <xsd:attribute name="value" type="xsd:decimal" />
    <xsd:attribute name="min" type="xsd:decimal" />
    <xsd:attribute name="max" type="xsd:int" />
    <xsd:attribute name="unit" type="xsd:string" />
  </xsd:complexType>
</xsd:element>
<xsd:element name="humidity">
  <xsd:complexType>
    <xsd:attribute name="value" type="xsd:int" />
    <xsd:attribute name="unit" type="xsd:string" />
  </xsd:complexType>
</xsd:element>
<xsd:element name="pressure">
  <xsd:complexType>
    <xsd:attribute name="value" type="xsd:int" />
    <xsd:attribute name="unit" type="xsd:string" />
  </xsd:complexType>
</xsd:element>
<xsd:element name="wind">
```



```
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="speed">
        <xsd:complexType>
          <xsd:attribute name="value" type="xsd:decimal" />
          <xsd:attribute name="name" type="xsd:string" />
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="direction">
        <xsd:complexType>
          <xsd:attribute name="value" type="xsd:decimal" />
          <xsd:attribute name="code" type="xsd:string" />
          <xsd:attribute name="name" type="xsd:string" />
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="clouds">
  <xsd:complexType>
    <xsd:attribute name="value" type="xsd:int" />
    <xsd:attribute name="name" type="xsd:string" />
  </xsd:complexType>
</xsd:element>
<xsd:element name="precipitation">
  <xsd:complexType>
    <xsd:attribute name="mode" type="xsd:string" />
  </xsd:complexType>
</xsd:element>
<xsd:element name="weather">
  <xsd:complexType>
    <xsd:attribute name="number" type="xsd:int" />
    <xsd:attribute name="value" type="xsd:string" />
    <xsd:attribute name="icon" type="xsd:string" />
  </xsd:complexType>
</xsd:element>
<xsd:element name="lastupdate">
  <xsd:complexType>
    <xsd:attribute name="value" type="xsd:dateTime" />
    <xsd:attribute name="unix" type="xsd:int" />
  </xsd:complexType>
</xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

```
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Diese Ausgabe sollte in einer Datei gespeichert werden, damit sie von `xjc` verwendet werden kann; nennen wir diese Datei etwa *weather.xsd*.

In Eclipse lässt sich xjc einfach aufrufen. Selektiere im Package-Explorer WEATER.XSD, im Kontextmenü GENERATE • JAXB CLASSES..., dann wähle das Java-Projekt aus, anschließend NEXT. Im folgenden Dialog gib als Paket »com.tutego.insel.xml.jaxb.weather« ein, dann klicke auf FINISH. Wichtig! Damit xjc gefunden wird, muss das JDK aktiviert sein, nicht das JRE; nur das JDK bringt das Werkzeug xjc mit.

Ein Aufruf von der Kommandozeile sieht so aus (Ausgabe wie sie Eclipse liefert):

```
$ xjc -p com.tutego.insel.xml.jaxb.weather weather.xsd
Ein Schema wird geparst ...
Ein Schema wird kompiliert ...
com\tutego\insel\xml\jaxb\weather\Cities.java
com\tutego\insel\xml\jaxb\weather\ObjectFactory.java
```

Das Tool generiert alle Typen für den komplexen Typ aus dem XML-Schema, um eine Paket-Annotation festmachen zu können, und `ObjectFactory`, die einfache Fabrikmethoden enthält, um die gemappten Objekte aufbauen zu können.

Dann kann ein Java-Programm den Service mit einer URL ansprechen und sich das Ergebnis-XML in eine JavaBean konvertieren lassen.

Listing 8.18 com/tutego/insel/xml/jaxb/Weather.java, main()

```
String url = "http://api.openweathermap.org/data/2.5/find?q=Frankfurt&units=metric&mode=xml";
Cities city = JAXB.unmarshal( url, Cities.class );
System.out.printf( "%.1f° C%n", city.getList().getItem().getTemperature().getValue() );
// z.B. 15,1° C
```

Konflikte in der Schema-Datei *

Leider haben viele XML-Schemas ein Problem, sodass sie nicht direkt vom Schema-Compiler verarbeitet werden können. Ein Beispiel zeigt das Dilemma:


```
<container>
 <head><content title="Titel"/></head>
 <body><content doc="doc.txt"/></body>
</container>
```

In der hierarchischen Struktur heißt das in `<head>` und `<body>` vorkommende XML-Element gleich, nämlich `content`. Die Schema-Datei kann widerspruchslos definieren, dass die beiden XML-Elemente gleich heißen, aber unterschiedliche Attribute erlauben, sozusagen das Head-Content und das Body-Content. Allein durch ihre Hierarchie, also dadurch, dass sie einmal unter `head` und einmal unter `body` liegen, sind sie eindeutig bestimmt. Der Schema-Compiler von Java bekommt aber Probleme, da er diese hierarchische Information in eine flache bringt. Er kann einfach eine Klasse `Head` und `Body` aufbauen, aber bei `<content>` steht er vor einem Problem. Da die Schema-Definitionen unterschiedlich sind, müssten zwei verschiedene Java-Klassen unter dem gleichen Namen `Content` generiert werden. Das geht nicht, und `xjc` bricht mit Fehlern ab.

Fehler dieser Art gibt es leider häufig, und sie sind der Grund, warum aus vielen Schemas nicht einfach JavaBeans generiert werden können. Erfolglos ohne weitere Einstellungen sind beispielsweise DocBook, Office Open XML, SVG, MathML und weitere. Doch was könnte die Lösung sein? `xjc` sieht Konfigurationsdateien vor, die das Mapping anpassen können. In diesen Mapping-Dokumenten identifiziert ein XPath-Ausdruck die problematische Stelle und gibt einen Substitutionstyp an. Die Dokumentation auf der JAXB-Homepage weist Interessierte in die richtige Richtung.

JAXB-Plugins

Auf der Webseite *http://java.net/projects/jaxb2-commons/* gibt es eine Reihe nützlicher zusätzlicher Plugins für JAXB. Zu ihnen gehören:

- **JAXB2 Basic Plugins**: Diese Plugin-Sammlung besteht aus Equals (erzeugt die `equals(…)`-Methode, wobei der Gleichheitstest über ein Equals-Strategie-Objekt austauschbar ist), HashCode (erzeugt `hashCode()`, wobei auch hier die Berechnung des Hashwerts über ein Strategie-Objekt erfolgt), ToString (erzeugt `toString()`, auch wieder über ein externes `ToStringStrategy`-Objekt), Copyable (erzeugt `copyTo(…)`, um Objektzustände in ein anderes typkompatibles Objekt zu kopieren, und auch `clone()`; arbeitet dabei mit `CopyStrategie`), Mergeable (realisiert `mergeFrom(…)`-Methoden mit `MergeStrategy`-Objekten, die Zustände eines anderen Objekts mit den eigenen vereinen), JAXBIndex (generiert eine *jaxb.index*-Datei, die alle generierten Schema-Klassen auflistet), Inheritance und AutoInheritance (erlauben JAXB-Beans, globale Elemente oder komplexe Typen von gewünschten Basisklassen zu erben oder Schnittstellen zu implementieren), Wildcard (spezifiziert, wie sich das Wildcard-Property verhalten soll), Setters (etwas andere Setter-Implementierung bei Sammlungen) und Simplify Plugin (komplexe Properties in Einzel-Properties aufspalten).
- **Camelcase Always Plugin**: Sind die Elementnamen groß geschrieben, so wird JAXB automatisch groß geschriebene Properties umsetzen, sodass etwa aus `NAME` der Setter/Getter

`setNAME(…)` und `getNAME()` entsteht. Das Plugin verhindert dies und nennt die Setter/Getter wie gewohnt `setName(…)` und `getName()`.

- **Value-Constructor Plugin**: Jede JavaBean bekommt von `xjc` nur einen Standard-Konstruktor. Das Plugin gibt einen weiteren Konstruktor hinzu, der alle Attribute direkt initialisiert.
- **Default Value Plugin**: Ein XML-Schema kann mit `defaultValue` vordefinierte Initialbelegungen für Attribute angeben. `xjc` ignoriert diese. Das Plugin wertet diese Vorbelegungen aus und initialisiert die Attribute der JavaBean gemäß den Werten.
- **Property Change Listener Injector Plugin**: Eine über `xjc` generierte JavaBean schreibt einen bei `setXXX(…)` übergebenen Wert direkt in das private Attribut durch. Mit dem Plugin wird ein `VetoableChangeListener` eingeführt, der gegen Wertänderungen votieren kann.
- **Fluent API Plugin**: Anstatt eine Bean nur mit Setter-Aufrufen zu initialisieren, etwa eine `Person` mit `setName(…)` und `setAge(…)`, generiert das Fluent API Plugin kaskadierbare Methoden, sodass im Beispiel unserer `Person` nur `new Person().withName(…).withAge(…)` zu schreiben ist.

8.5 Serielle Verarbeitung mit StAX

Mit der Pull-API *StAX* lassen sich XML-Dokumente sehr performant ablaufen, jedoch nicht ändern.

Die allgemeine Vorgehensweise zum Parsen eines XML-Dokuments ist folgende:

1. Erzeuge eine `XMLInputFactory`.
2. Erzeuge den passenden Parser.
3. Wähle `XMLStreamReader` für die Cursor-Verarbeitung oder `XMLEventReader` für die Iterator-Verarbeitung.
4. Erfrage mit `next()` die nächste Komponente des XML-Dokuments.
5. Ermittle den Typ der Komponente, und verarbeite ihn.

8.5.1 Unterschiede der Verarbeitungsmodelle

Die Unterschiede zwischen der Cursor- und der Iterator-Verarbeitung sind auf den ersten Blick nicht eindeutig. Beide Verarbeitungsmodelle bieten ähnliche Methoden, und die Verarbeitung der Inhalte ist auch sehr ähnlich. Der wesentliche Unterschied ist die Art und Weise, wie die Komponenten des XML-Dokuments geliefert werden:

- Bei der *Cursor-Verarbeitung* wird die Komponente direkt mit dem Parser-Objekt verarbeitet. Hier ist die zentrale Klasse der `XMLStreamReader`, mit dem auch auf die Inhalte der XML-Datei zugegriffen wird. Da diese Klasse ebenso verwendet wird, um auf das nächste Element der XML-Datei zugreifen zu können, steht zu einem Zeitpunkt immer nur eine Komponente des XML-Dokuments zur Verfügung. Der Vorteil ist die hohe Effizienz, da bei der Verarbeitung keine neuen Objekte erzeugt werden. Ein XML-Parser ist ein Zustandsautomat, und Methoden führen den Automaten von einem Zustand in den nächsten.

- Bei der *Iterator-Verarbeitung* wird ein `XMLEvent`-Objekt geliefert, das anderen Methoden übergeben und in einer Datenstruktur gespeichert werden kann.

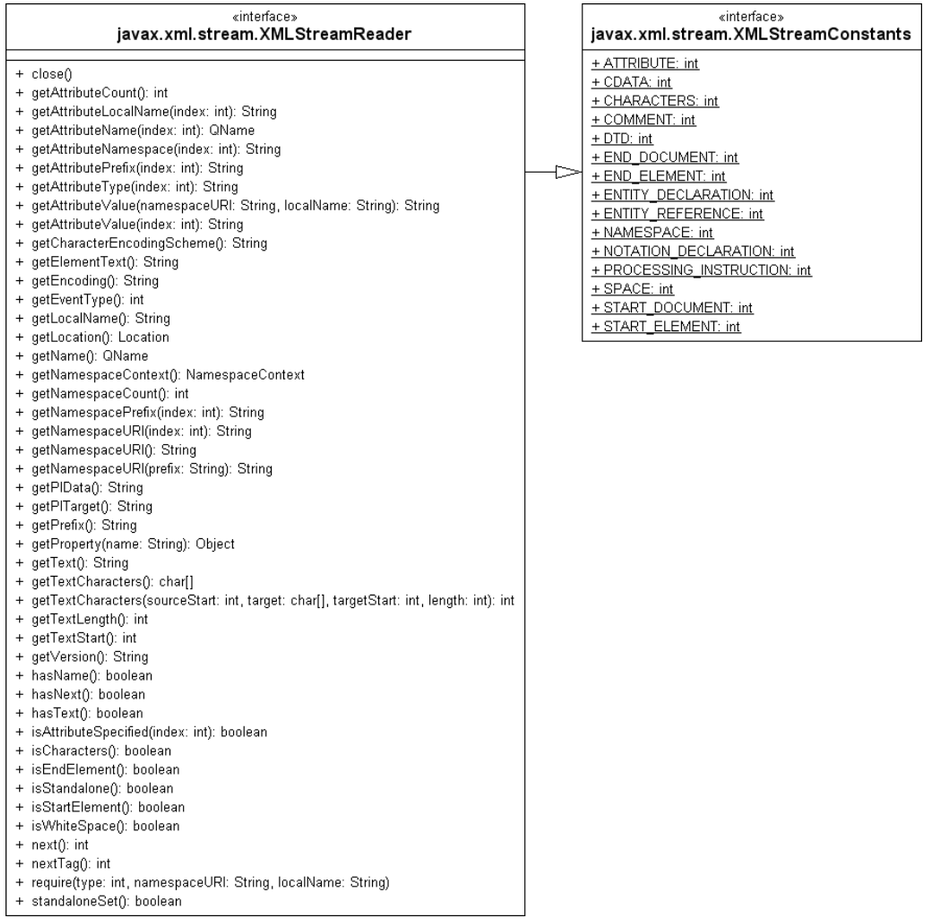

Abbildung 8.2 XMLStreamReader-Objekte erzeugen Ereignisse.

StAX ist eine symmetrische API, was bedeutet, dass es Klassen zum Lesen und auch zum Schreiben von XML-Dokumenten gibt. So wie es für das Lesen die Prinzipien *Cursor* und *Iterator* gibt, so bietet die StAX-API die Typen `XMLStreamWriter` und `XMLEventWriter`. Damit ist es möglich, Elemente, die über die Reader gelesen werden, an die Writer zu übergeben und damit Änderungen an den Inhalten zu schreiben.

8.5.2 XML-Dateien mit dem Cursor-Verfahren lesen

Zunächst muss ein Parser erzeugt werden, der das XML-Dokument verarbeiten soll:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader parser  = factory.createXMLStreamReader( input );
```

Der Parser iteriert über die XML-Datei mit einer Tiefensuche und liefert beim Verarbeiten eine Reihe von Events, die den Typ des XML-Elements anzeigen. Die Eventtypen sind ganzzahlige Werte und als Konstanten in der Klasse `XMLStreamConstants` festgelegt. Der Parser liefert die folgenden Elemente:

XMLStreamConstants-Konstante	Beschreibung
`START_DOCUMENT`	Zeigt den Beginn der XML-Datei an. Bei diesem Event können Eigenschaften wie das Encoding des Dokuments ermittelt werden.
`END_DOCUMENT`	Zeigt das Ende des Dokuments an. Hier steht nur die Methode `close()` zum Schließen der Eingabe zur Verfügung.
`START_ELEMENT`	Zeigt an, dass ein Element beginnt. Die Attribute und der Namensraum eines Elements können hier ausgewertet werden.
`END_ELEMENT`	Zeigt an, dass das Ende eines Elements erreicht ist.
`CHARACTERS`	Zeigt Text innerhalb von Elementen. Text kann auf Whitespace getestet werden.
`ENTITY_REFERENCE`	Zeigt Entitäten in der XML-Datei an. Üblicherweise werden Entitäten zuerst aufgelöst und dann als `CHARACTERS`-Event geliefert.
`DTD`	Zeigt die DTD als String, sodass es möglich wird, auf Teile der DTD zuzugreifen.
`COMMENT`	Zeigt Kommentare in der XML-Datei an.
`PROCESSING_INSTRUCTION`	Zeigt Verarbeitungsanweisungen wie Stylesheet-Angaben.

Tabelle 8.14 Event-Typen und was sie bedeuten

Die Events `ATTRIBUTE` und `NAMESPACE` liefert der Parser nur in Ausnahmefällen. Inhalte von Attributen sowie die Namensraumdaten lassen sich beim Event `START_ELEMENT` erfragen.

Passend zum Event sind verschiedene Methodenaufrufe gültig, etwa `getAttributeCount()` im Fall eines Elements, das die Anzahl der Attribute liefert. Mit einer Schleife und einer `switch`-Anweisung lassen sich die Inhalte der XML-Datei dann einfach auswerten:

Listing 8.19 com/tutego/insel/xml/stax/XMLStreamReaderDemo.java, main() Ausschnitt

```
try ( Reader in =
 Files.newBufferedReader( Paths.get( "party.xml" ), StandardCharsets.UTF_8 ) ) {
XMLStreamReader parser = XMLInputFactory.newInstance().createXMLStreamReader( in );
StringBuilder spacer = new StringBuilder();

while ( parser.hasNext() ) {
  System.out.println( "Event: " + parser.getEventType() );

  switch ( parser.getEventType() ) {
    case XMLStreamConstants.START_DOCUMENT:
      System.out.println( "START_DOCUMENT: " + parser.getVersion() );
      break;

    case XMLStreamConstants.END_DOCUMENT:
      System.out.println( "END_DOCUMENT: " );
      parser.close();
      break;

    case XMLStreamConstants.NAMESPACE:
      System.out.println( "NAMESPACE: " + parser.getNamespaceURI() );
      break;
  break;

case XMLStreamConstants.START_ELEMENT:
  spacer.append( "  " );
  System.out.println( spacer + "START_ELEMENT: " + parser.getLocalName() );

  // Der Event XMLStreamConstants.ATTRIBUTE wird nicht geliefert!
  for ( int i = 0; i < parser.getAttributeCount(); i++ )
    System.out.println( spacer + "  Attribut: "
                        + parser.getAttributeLocalName( i )
                        + " Wert: " + parser.getAttributeValue( i ) );
  break;

case XMLStreamConstants.CHARACTERS:
  if ( ! parser.isWhiteSpace() )
    System.out.println( spacer + "  CHARACTERS: " + parser.getText() );
  break;

case XMLStreamConstants.END_ELEMENT:
  System.out.println( spacer + "END_ELEMENT: " + parser.getLocalName() );
  spacer.delete( (spacer.length() - 2), spacer.length() );
  break;
```

```
    default:
      break;
  }
  parser.next();
}
parser.close();
}
catch ( IOException | XMLStreamException e ) {
  e.printStackTrace();
}
```

Dieses Beispiel demonstriert das Lesen einer XML-Datei mit dem Cursor-Verfahren. Der folgende Abschnitt 8.5.3 zeigt die Verarbeitung mit dem Iterator-Verfahren.

[+]

Hinweis

Der `XMLStreamReader` liefert beim Parsen keinen Typ `XMLStreamConstants.ATTRIBUTE`. Dieses Event kann nur im Zusammenhang mit XPath auftreten, wenn der Ausdruck ein Attribut als Rückgabe liefert. Beim Parsen von XML-Dokumenten werden Attribute über ihre Elemente geliefert.

`abstract class javax.xml.stream.`**`XMLInputFactory`**

- `static XMLInputFactory newInstance()`
  Liefert ein Exemplar der Fabrik `XMLInputFactory`. Aus dem Objekt erfolgt als Nächstes üblicherweise ein Aufruf von `createXMLEventReader(…)`.
- `abstract XMLStreamReader createXMLStreamReader(InputStream stream)`
- `abstract XMLStreamReader createXMLStreamReader(InputStream stream, String encoding)`
- `abstract XMLStreamReader createXMLStreamReader(Reader reader)`
- `abstract createXMLStreamReader(Source source)`
  Liefert einen `XMLStreamReader`, der aus unterschiedlichen Quellen liest.

`interface javax.xml.stream.`**`XMLStreamReader`**
`extends XMLStreamConstants`

- `boolean hasNext()`
  Sagt, ob es noch ein neues Parse-Event gibt.
- `int getEventType()`
  Liefert den Typ des Parse-Events, so wie in `XMLStreamConstants` deklariert. Die Schnittstelle `XMLStreamReader` erweitert `XMLStreamConstants`.

- `int next()`
  Parst das nächste Element und liefert das nächste Parse-Event.
- Die Javadoc listet die `getXXX()`-Methoden auf, die alle Eigenschaften wie Namensraum, Attribute usw. des Elements liefert. Die nächste Tabelle zeigt, welche Methoden auf welchen Zuständen erlaubt sind:

Event-Typ	Erlaubte Methoden
auf allen Zuständen	`hasNext()`, `require()`, `close()`, `getNamespaceURI()`, `isStartElement()`, `isEndElement()`, `isCharacters()`, `isWhiteSpace()`, `getNamespaceContext()`, `getEventType()`, `getLocation()`, `hasText()`, `hasName()`, `getProperty()`
`START_ELEMENT`	`next()`, `getName()`, `getLocalName()`, `hasName()`, `getPrefix()`, `getAttributeXXX()`, `isAttributeSpecified()`, `getNamespaceXXX()`, `getElementText()`, `nextTag()`
`ATTRIBUTE`	`next()`, `nextTag()`, `getAttributeXXX()`, `isAttributeSpecified()`
`NAMESPACE`	`next()`, `nextTag()`, `getNamespaceXXX()`
`END_ELEMENT`	`next()`, `getName()`, `getLocalName()`, `hasName()`, `getPrefix()`, `getNamespaceXXX()`, `nextTag()`
`CHARACTERS`, `CDATA`, `COMMENT`, `SPACE`	`next()`, `getTextXXX()`, `nextTag()`
`START_DOCUMENT`	`next()`, `getEncoding()`, `getVersion()`, `isStandalone()`, `standaloneSet()`, `getCharacterEncodingScheme()`, `nextTag()`
`PROCESSING_INSTRUCTION`	`next()`, `getPITarget()`, `getPIData()`, `nextTag()`
`ENTITY_REFERENCE`	`next()`, `getLocalName()`, `getText()`, `nextTag()`
`DTD`	`next()`, `getText()`, `nextTag()`
`END_DOCUMENT`	`close()`

Tabelle 8.15 Erlaubte Methoden der Event-Typen

8.5.3 XML-Dateien mit dem Iterator-Verfahren verarbeiten *

Die Verarbeitung mit der Iterator-Variante der StAX-API ist ein wenig komplizierter, aber auch viel flexibler. Es wird nicht direkt mit dem allgemeinen Parser-Objekt auf die Daten zugegriffen, sondern es wird bei jeder Iteration ein `XMLEvent`-Objekt erzeugt. Mit diesem Objekt kann der Typ des Events ermittelt und ganz ähnlich wie bei der Cursor-API ausgewertet werden.

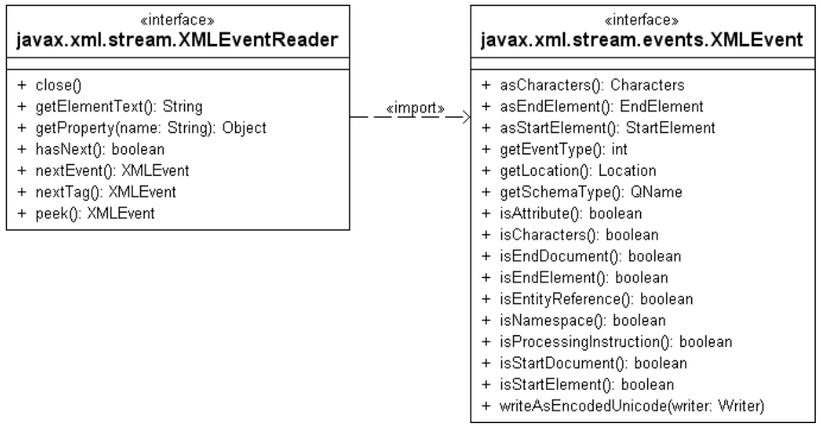

Abbildung 8.3 Klassendiagramm für XMLEventReader und erzeugte Events

Am Anfang wird ein Parser vom Typ `XMLEventReader` erzeugt, und in einer Schleife werden die einzelnen Komponenten ausgewertet:

Listing 8.20 com/tutego/insel/xml/stax/XMLEventReaderDemo.java, main()

```
try ( Reader in = Files.newBufferedReader( Paths.get( "party.xml" ),
                                           StandardCharsets.UTF_8 ) ) {
  XMLInputFactory factory = XMLInputFactory.newInstance();
  XMLEventReader parser = factory.createXMLEventReader( in );

  StringBuilder spacer = new StringBuilder();
  while ( parser.hasNext() ) {
    XMLEvent event = parser.nextEvent();

    switch ( event.getEventType() ) {
      case XMLStreamConstants.START_DOCUMENT:
        System.out.println( "START_DOCUMENT:" );
        break;
      case XMLStreamConstants.END_DOCUMENT:
        System.out.println( "END_DOCUMENT:" );
        parser.close();
        break;
      case XMLStreamConstants.START_ELEMENT:
        StartElement element = event.asStartElement();
        System.out.println( spacer.append("  ")
                            + "START_ELEMENT: "
                            + element.getName() );
```



```
        for ( Iterator<?> attributes = element.getAttributes();
              attributes.hasNext(); ) {
          Attribute attribute = (Attribute) attributes.next();
          System.out.println( spacer + "  Attribut: "
                              + attribute.getName() + " Wert: "
                              + attribute.getValue() );
        }
        break;
      case XMLStreamConstants.CHARACTERS:
       Characters characters = event.asCharacters();
        if ( ! characters.isWhiteSpace() )
          System.out.println( spacer
                              + "  CHARACTERS: "
                              + characters.getData() );
        break;
      case XMLStreamConstants.END_ELEMENT:
        System.out.println( spacer
                            + "END_ELEMENT: "
                            + event.asEndElement().getName() );
        spacer.delete( (spacer.length() - 2), spacer.length() );
        break;
      case XMLStreamConstants.ATTRIBUTE:
        break;

      default :
        break;
    }
  }
  parser.close();
}
catch ( IOException | XMLStreamException e ) {
  e.printStackTrace();
}
```

Diese Form der Verarbeitung sieht auf den ersten Blick komplizierter aus, bietet aber zusätzliche Möglichkeiten, weil die erzeugten Objekte für die weitere Verarbeitung zur Verfügung stehen.

`abstract class javax.xml.stream.XMLInputFactory`

- `abstract XMLEventReader createXMLEventReader(InputStream stream)`
- `abstract XMLEventReader createXMLEventReader(InputStream stream, String encoding)`
- `abstract XMLEventReader createXMLEventReader(Reader reader)`
- `abstract XMLEventReader createXMLEventReader(Source source)`

- `abstract XMLEventReader createXMLEventReader(XMLStreamReader reader)`
  Liefert `XMLEventReader`, der die Eingabe aus unterschiedlichen Quellen liest.

Der `XMLEventReader` implementiert nicht `AutoCloseable`, bietet aber eine `close()`-Methode. Unüblicherweise schließt sie aber nicht die Ressource, auf der sie arbeitet. Explizit schließt in unserem Beispiel das `try` mit Ressourcen die Datei.

8.5.4 Mit Filtern arbeiten *

Mithilfe von Filtern gibt es die Möglichkeit, nur Teile eines XML-Dokuments zu parsen. Diese Filter werden durch die Implementierung einer der Schnittstellen `javax.xml.stream.EventFilter` (für die XML-Events) oder `javax.xml.stream.StreamFilter` (für die `XMLStreamReader`) programmiert. Es muss lediglich die Methode `accept(XMLEvent)` implementiert und ein `boolean`-Wert zurückgegeben werden. Als Parameter erwartet diese Methode entweder einen `javax.xml.stream.events.XMLEvent` bei der Iterator-Variante oder einen `XMLStreamReader` bei der Cursor-Variante. Dazu ein Beispiel: Ein Filter soll für die Iterator-Variante die schließenden Tags auslassen:

Listing 8.21 com/tutego/insel/xml/stax/PartyEventFilter.java

```
package com.tutego.insel.xml.stax;

import javax.xml.stream.EventFilter;
import javax.xml.stream.events.XMLEvent;

public class PartyEventFilter implements EventFilter {
  @Override public boolean accept( XMLEvent event ) {
    return ! event.isEndElement();
  }
}
```

Der Filter wird beim Erzeugen des Parsers mit der `XMLInputFactory` und dem vorhandenen `XMLEventReader` erzeugt. Dazu ein Beispiel zur Erzeugung des Parsers mit dem Event-Filter:

```
XMLEventReader filteredParser = factory.createFilteredReader(
                                parser, new PartyEventFilter() );
```

Dieses Verfahren der Dekoration wird in ähnlicher Form bei Streams verwendet.

Das Erzeugen eines Parsers mit einem Filter für die Cursor-Variante funktioniert analog. Mit Filtern bietet die API eine einfache Lösung, wenn nur bestimmte Teile des XML-Dokuments verarbeitet werden sollen.

`abstract class javax.xml.stream.`**`XMLInputFactory`**

- `abstract XMLEventReader createFilteredReader(XMLEventReader reader, EventFilter filter)`

- `abstract XMLStreamReader createFilteredReader(XMLStreamReader reader, StreamFilter filter)`
  Liefert `XMLEventReader` oder `XMLStreamReader` mit einem Filter.

`interface javax.xml.stream.`**`EventFilter`**

- `boolean accept(XMLEvent event)`
  Liefert `true`, wenn das Ereignis in den Ergebnisstrom soll.

`interface javax.xml.stream.`**`StreamFilter`**

- `boolean accept(XMLStreamReader reader)`
  Liefert `false`, wenn der `XMLStreamReader` in einem Zustand ist, bei dem das Element ignoriert werden soll.

8.5.5 XML-Dokumente schreiben

Im Gegensatz zu DOM-orientierten APIs, bei denen das gesamte Dokument im Speicher vorliegt und verändert werden kann, ist es bei StAX nicht möglich, die vorhandene XML-Datei zu verändern. Es ist aber trotzdem möglich, XML zu schreiben. Auch hier wird zwischen der Cursor- und der Iterator-Variante unterschieden. Bei der Iterator-Variante werden Event-Objekte geschrieben, die entweder aus einem gelesenen XML-Dokument stammen oder mit einer `XMLEventFactory` erzeugt werden. Bei der Cursor-Variante wird mit einem `XMLStreamWriter` die XML-Komponente direkt erzeugt und geschrieben. In beiden Fällen wird über die `XMLOutputFactory` ein passender Writer erzeugt. Die Reihenfolge, in der die Komponenten geschrieben werden, entscheidet über den Aufbau des zu erzeugenden XML-Dokuments.

XMLStreamWriter

Zuerst zeigen wir, wie mit der Cursor-Variante eine XML-Datei geschrieben werden kann. Dazu erzeugen wir mit der `XMLOutputFactory` einen `XMLStreamWriter`, der die Elemente und Attribute direkt in eine XML-Datei schreibt:

Listing 8.22 com/tutego/insel/xml/stax/XMLStreamWriterDemo.java, Ausschnitt main()

```
XMLOutputFactory factory = XMLOutputFactory.newInstance();
try ( OutputStream stream =
  Files.newOutputStream( Paths.get( "writenParty.xml" ) ) ) {
XMLStreamWriter writer = factory.createXMLStreamWriter( stream );
// Der XML-Header wird erzeugt
writer.writeStartDocument();
// Zuerst wird das Wurzelelement mit Attribut geschrieben
  writer.writeStartElement( "party" );
    writer.writeAttribute( "datum", "31.12.2012" );
```

```
    // Unter diesem Element das Element gast mit einem Attribut erzeugen
    writer.writeStartElement( "gast" );
      writer.writeAttribute( "name", "Albert Angsthase" );
    writer.writeEndElement();
  writer.writeEndElement();
writer.writeEndDocument();
writer.close();
}
```

Beim Schreiben werden zuvor keine speziellen Objekte in einem XML-Baum erzeugt, sondern die Elemente werden direkt geschrieben. Der große Vorteil ist, dass das Schreiben sehr performant ist und die Größe der XML-Ausgabe beliebig sein kann.

XMLEventWriter *

Das Schreiben von XML-Dokumenten mit dem `XMLEventWriter` erfolgt in drei Stufen:

1. Von der `XMLOutputFactory` wird ein Objekt vom Typ `XMLEventWriter` erfragt. In den `XMLEventWriter` werden dann die Ereignisobjekte geschrieben.
2. Für das Erzeugen der Event-Objekte wird eine `XMLEventFactory` benötigt. Mit ihr lassen sich neue `XMLEvent`-Objekte erzeugen und irgendwo speichern.
3. Die `XMLEvent`-Objekte werden geschrieben.

Der zentrale Unterschied zwischen dem `XMLStreamWriter` und `XMLEventWriter` ist also, dass beim `XMLEventWriter` erst die `XMLEvent`-Objekte erzeugt werden – in beliebiger Reihenfolge – und dass sie dann in den `XMLEventWriter` kommen; die Reihenfolge beim Erzeugen hat keinen Einfluss auf die Reihenfolge in der späteren Ausgabe:

Listing 8.23 com/tutego/insel/xml/stax/XMLEventWriterDemo.java, main() Teil 1

```
try ( Writer out =
 Files.newBufferedWriter( Paths.get( "writenParty.xml" ), StandardCharsets.UTF_8 ) ) {
XMLEventWriter writer = XMLOutputFactory.newInstance().createXMLEventWriter( out );
XMLEventFactory eventFactory = XMLEventFactory.newInstance();
XMLEvent header = eventFactory.createStartDocument();
XMLEvent startRoot = eventFactory.createStartElement( "", "", "party" );
XMLEvent datumAttribut = eventFactory.createAttribute( "datum", "31.12.2012" );
XMLEvent endRoot = eventFactory.createEndElement( "", "", "party" );

XMLEvent startGast = eventFactory.createStartElement( "", "", "gast" );
XMLEvent name = eventFactory.createAttribute( "name",  "Albert Angsthase" );
XMLEvent endGast = eventFactory.createEndElement( "", "", "gast" );
XMLEvent endDocument = eventFactory.createEndDocument();
```

Zuerst werden für das Wurzelelement das öffnende und schließende Tag sowie das Attribut `datum` erzeugt. Hierfür wird die Methode `createStartElement(…)` verwendet.

Um die Elemente zu schreiben, werden sie dem `XMLEventWriter` übergeben. Hier entscheidet die Reihenfolge über den Aufbau der XML-Datei:

Listing 8.24 com/tutego/insel/xml/stax/XMLEventWriterDemo.java, main() Teil 2

```
writer.add( header );
writer.add( startRoot );
    writer.add( datumAttribut );
    writer.add( startGast );
      writer.add( name );
    writer.add( endGast );
  writer.add( endRoot );
writer.add( endDocument );
writer.close();
}
catch ( IOException | XMLStreamException e ) {
  e.printStackTrace();
}
```

In diesem Beispiel wurde gezeigt, wie Events erzeugt werden können und wie sie geschrieben werden. Der Strom muss explizit mit `close()` vom `XMLEventWriter` geschlossen werden, wobei `XMLEventWriter` nicht `AutoCloseable` ist. Die Ressource selbst (also der `Writer` in unserem Fall) muss ebenfalls geschlossen werden, was aber unser `try` mit Ressourcen übernimmt.

Das Schreiben von Elementen aus einer Eingabe funktioniert analog. Falls sich Elemente wiederholen oder aus einer anderen Quelle stammen (etwa ein `XMLEvent`, das serialisiert vom Netzwerk kommt), können sie direkt in den `XMLEventWriter` geschrieben werden. In dem Fall ist die Iterator-Variante flexibler als die Cursor-Variante. Diese Flexibilität wird durch einen etwas höheren Aufwand erkauft.

Hinweis

Die Ausgabe ist weder beim `XMLStreamWriter` noch beim `XMLEventWriter` formatiert oder eingerückt, und die XML-Elemente stehen einfach hintereinander:

```
<?xml version="1.0" ?><party datum="31.12.2012"><gast name="Albert Angsthase">
<getraenk>Wein</getraenk><getraenk>Bier</getraenk><zustand ledig="true"
nuechtern="true"/></gast></party>
```

Besteht die Anforderung, dass die XML-Ausgabe eingerückt ist, kann Java SE standardmäßig nichts machen, und es muss auf externe Hilfsklassen zurückgegriffen werden. Die StAX-Utility-Sammlung unter *https://java.net/projects/stax-utils/pages/Utilities* bietet den `javanet.staxutils.IndentingXMLEventWriter` bzw. `IndentingXMLStreamWriter`, dessen Nutzung so aussieht:

```
XMLStreamWriter writer =
  XMLOutputFactory.newInstance().createXMLStreamWriter( out );
```

```
writer = new IndentingXMLStreamWriter( writer );
writer.writeStartDocument();
```

Der `IndentingXMLStreamWriter` realisiert das Dekorator-Pattern.

Zusammenfassung

Wir haben gesehen, wie mit der StAX-API XML gelesen und geschrieben werden kann, welche Unterschiede zwischen der Cursor- und der Iterator-Variante bestehen und welche Filter für die Eingabe zur Verfügung stehen. Grundsätzlich ist die Iterator-Variante die flexiblere Lösung und in den meisten Fällen performant genug. Sie ist in jedem Fall performanter als eine DOM-basierte Lösung, wenn nicht die gesamte XML-Struktur im Speicher benötigt wird. Die Cursor-Variante sollte gewählt werden, wenn hohe Verarbeitungsgeschwindigkeit und geringer Speicherverbrauch Priorität haben. Diese Variante ist insbesondere für Endgeräte mit wenig Speicher und geringer Rechenleistung die bessere Wahl.

Die Anwendungsgebiete der StAX-API sind die gleichen wie die der SAX-API, weil die Vorteile beider Verfahren gute Performance und geringer Speicherverbrauch sind. Für die meisten Programmierer ist diese Form der Verarbeitung einfacher als die SAX-Variante, weil der XML-Inhalt direkt gelesen wird. SAX hat den Vorteil, dass es weit verbreitet ist und in vielen Programmiersprachen zur Verfügung steht. Wir stellen SAX im folgenden Abschnitt kurz vor.

8.6 Serielle Verarbeitung von XML mit SAX *

Die Verarbeitung von XML-Dateien mit SAX ist vor dem Erscheinen von StAX die schnellste und speicherschonendste Methode gewesen. Der Parser liest die XML-Datei seriell und ruft für jeden Bestandteil der XML-Datei eine spezielle Methode auf. Der Nachteil ist, dass immer nur ein kleiner Bestandteil einer XML-Datei betrachtet wird und nicht die gesamte Struktur zur Verfügung steht.

8.6.1 Schnittstellen von SAX

Die bei der Verarbeitung mit SAX anfallenden Ereignisse sind in verschiedenen Schnittstellen festgelegt. Die wichtigste Schnittstelle ist `org.xml.sax.ContentHandler`. Die Schnittstelle legt die wichtigsten Operationen für die Verarbeitung fest, denn die später realisierten Methoden ruft der Parser beim Verarbeiten der XML-Daten auf.

Die Klasse `org.xml.sax.helpers.DefaultHandler` ist eine leere Implementierung aller Operationen aus `ContentHandler`. Zusätzlich implementiert `DefaultHandler` die Schnittstellen `DTDHandler`, `EntityResolver` und `ErrorHandler`. Auf diese Schnittstellen wird hier nicht näher eingegangen.

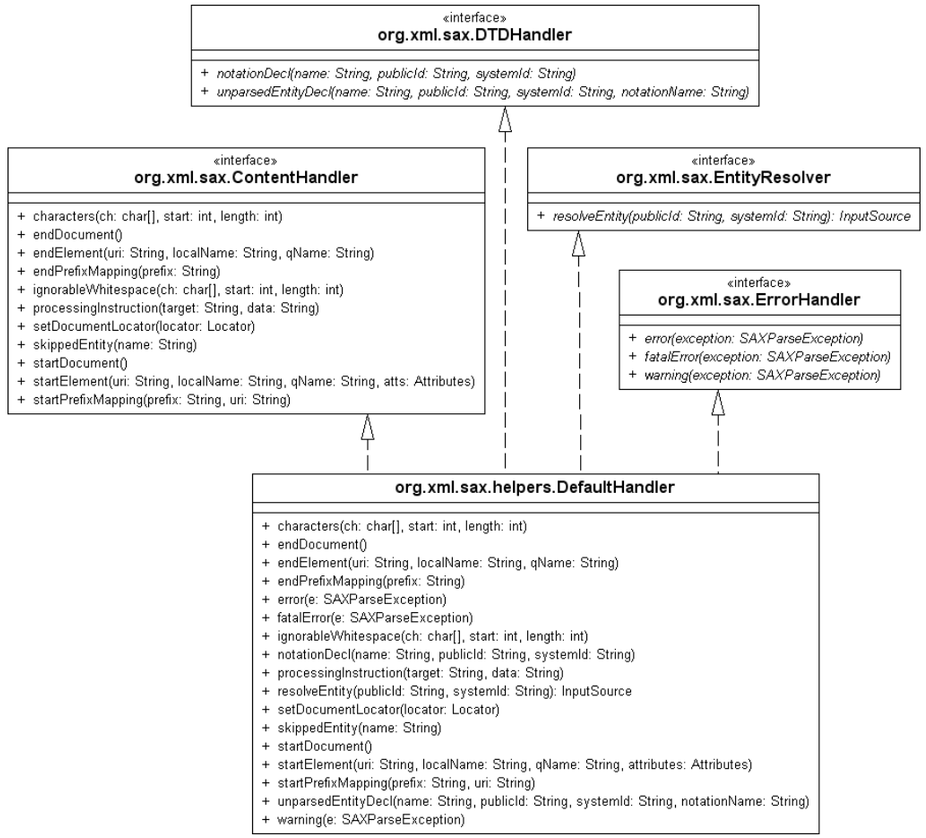

Abbildung 8.4 Vererbungsbeziehung der SAX-Handler

8.6.2 SAX-Parser erzeugen

Um zum Parsen einer Datei zu kommen, führt der Weg über zwei Fabrikmethoden:

Listing 8.25 com/tutego/insel/xml/sax/SaxParty.java, main()

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
DefaultHandler handler = new PartyHandler();
try ( InputStream in = Files.newInputStream( Paths.get( "party.xml" ) ) ) {
  saxParser.parse( in, handler );
}
```

Mit dem `SAXParser` erledigt `parse(…)` das Einlesen. Die Methode benötigt die Datei und eine Implementierung der Callback-Methoden, die wir als `PartyHandler` bereitstellen.

8.6.3 Operationen der Schnittstelle ContentHandler

`DefaultHandler` ist eine Klasse, die alle Operationen aus `EntityResolver`, `DTDHandler`, `ContentHandler` und `ErrorHandler` leer implementiert. Unsere Unterklasse `PartyHandler` erweitert die Klasse `DefaultHandler` und überschreibt interessantere Methoden, die wir mit Leben füllen wollen:

Listing 8.26 com/tutego/insel/xml/sax/PartyHandler.java, Teil 1

```
package com.tutego.insel.xml.sax;

import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;

class PartyHandler extends DefaultHandler {
```

Beim Start und Ende des Dokuments ruft der Parser die Methoden `startDocument()` und `endDocument()` auf. Unsere überschriebenen Methoden geben nur eine kleine Meldung auf dem Bildschirm aus:

Listing 8.27 com/tutego/insel/xml/sax/PartyHandler.java, Teil 2

```
  @Override
  public void startDocument() {
    System.out.println( "Document starts." );
  }

  @Override
  public void endDocument() {
    System.out.println( "Document ends." );
  }
```

Sobald der Parser ein Element erreicht, ruft er die Methode `startElement(…)` auf. Der Parser übergibt der Methode die Namensraumadresse, den lokalen Namen, den qualifizierenden Namen und Attribute:

Listing 8.28 com/tutego/insel/xml/sax/PartyHandler.java, Teil 3

```
  @Override
  public void startElement( String namespaceURI, String localName,
                            String qName, Attributes atts ) {
    System.out.println( "namespaceURI: " + namespaceURI );
    System.out.println( "localName: " + localName );
    System.out.println( "qName: " + qName );

    for ( int i = 0; i < atts.getLength(); i++ )
      System.out.printf( "Attribut no. %d: %s = %s%n", i,
                         atts.getQName( i ), atts.getValue( i ) );
  }
```


Unsere Methode gibt alle notwendigen Informationen eines Elements aus. Falls kein spezieller Namensraum vergeben ist, sind die Strings `namespaceURI` und `localName` leer. Der String `qName` ist immer gefüllt. Die Attribute enthält der Container `Attributes`. Das schließende Tag eines Elements verarbeitet die Methode `endElement(String namespaceURI, String localName, String qName)`. Bis auf die Attribute sind auch bei dem schließenden Tag alle Informationen für die Identifizierung des Elements vorhanden. Auch hier sind die Strings `namespaceURI` und `localName` leer, falls kein spezieller Namensraum verwendet wird.

Den Inhalt eines Elements verarbeitet unsere letzte Methode, `characters()`:

Listing 8.29 com/tutego/insel/sax/PartyHandler.java, Teil 4

```
  @Override
  public void characters( char[] ch, int start, int length ) {
    System.out.println( "Characters:" );

    for ( int i = start; i < (start + length); i++ )
      System.out.printf( "%1$c (%1$x) ", (int) ch[i] );

    System.out.println();
  }
}
```

Es ist nicht festgelegt, ob der Parser den Text in einem Stück liefert oder in kleinen Stücken.[4] Zur besseren Sichtbarkeit geben wir neben dem Zeichen selbst auch seinen Hexadezimalwert aus. So beginnt die Ausgabe mit den Zeilen:

```
Document starts.
namespaceURI:
localName:
qName: party
Attribut no. 0: datum = 31.12.2012
Characters:


 (a)
 (a)   (20)   (20)   (20)
namespaceURI:
```

4 Die Eigenschaft nennt sich *Character Chunking*: *http://www.tutego.de/blog/javainsel/2007/01/character-%E2%80%9Echunking%E2%80%9C-bei-sax/*

```
localName:
qName: gast
Attribut no. 0: name = Albert Angsthase
Characters:

 (a)   (20)   (20)   (20)   (20)   (20)   (20)
namespaceURI:
localName:
qName: getraenk
Characters:
W (57) e (65) i (69) n (6e)
Characters:

 (a)   (20)   (20)   (20)   (20)   (20)   (20)
namespaceURI:
localName:
qName: getraenk
Characters:
B (42) i (69) e (65) r (72)
```

8.6.4 ErrorHandler und EntityResolver

Immer dann, wenn der Parser einen Fehler melden muss, ruft er die im `ErrorHandler` deklarierten Operationen auf:

```
interface org.xml.sax.ErrorHandler
```

- `void warning(SAXParseException exception)`
- `void error(SAXParseException exception)`
- `void fatalError(SAXParseException exception)`

Da der `DefaultHandler` die Methoden `warning(…)` und `error(…)` leer implementiert, fällt kein Fehler wirklich auf; nur bei `fatalError(…)` leitet die Methode den empfangenen Fehler mit `throw` weiter. Das heißt aber auch, dass zum Beispiel schwache Validierungsfehler nicht auffallen. Eine Implementierung kann aber wie folgt aussehen:

```
public void error( SAXParseException e ) throws SAXException {
  throw new SAXException( saxMsg(e) );
}
private String saxMsg( SAXParseException e ) {
  return   "Line: " + e.getLineNumber() + ", Column: "
         + e.getColumnNumber() + ", Error: " + e.getMessage();
}
```


Die Klasse `DefaultHandler` implementiert ebenso die Schnittstelle `EntityResolver`, aber auch hier einfach die eine Methode `InputSource resolveEntity (String publicId, String systemId)` mit einem `return null`. Das heißt, die Standardimplementierung löst keine Entities auf. Eigene Implementierungen sehen meist im Kern so aus:

```
InputStream stream = MyEntityResolver.class.getResourceAsStream( dtd );
return new InputSource( new InputStreamReader( stream ) );
```

Die Variable `dtd` ist mit dem Pfadnamen einer DTD belegt, die im Klassenpfad liegen muss.

8.7 XML-Dateien mit JDOM verarbeiten

Über JDOM lassen sich die XML-formatierten Dateien einlesen, manipulieren und dann wieder schreiben. Mit einfachen Aufrufen lässt sich ein Dokument im Speicher erstellen. Zur internen JDOM-Repräsentation werden einige Java-typische Features verwendet, beispielsweise die Collection-API zur Speicherung, Reflection oder schwache Referenzen. Die Nutzung der Collection-API ist ein Vorteil, der unter dem herkömmlichen DOM nicht zum Tragen kommt. Durch JDOM können mit dem `new`-Operator auch Elemente und Attribute einfach erzeugt werden. Es gibt spezielle Klassen für das Dokument, nämlich Elemente, Attribute und Kommentare. Es sind keine Fabrikschnittstellen, die konfiguriert werden müssen, sondern alles wird direkt erzeugt.

Die Modelle StAX, SAX oder DOM liegen eine Ebene unter JDOM, denn sie dienen als Ausgangspunkt zum Aufbau eines JDOM-Baums. Das heißt, dass ein vorgeschalteter SAX- oder StAX-Parser (bei JDOM *Builder* genannt) die JDOM-Baumstruktur im Speicher erzeugt. Die Bibliothek bietet daher eine neutrale Schnittstelle für diverse Parser, um die Verarbeitung der XML-Daten so unabhängig von den Implementierungen wie möglich zu machen. JDOM unterstützt dabei aktuelle Standards wie DOM Level 3, SAX 2.0 oder XML Schema. Wenn es nötig wird, DOM oder SAX zu unterstützen, bieten Schnittstellen diesen Einstieg an.

Mit JDOM wird auch eine interne Datenstruktur der XML-Datei erzeugt. Dadurch kann jederzeit auf alle Elemente der XML-Datei zugegriffen werden. Da JDOM Java-spezifische Datenstrukturen verwendet, ist die Verarbeitung effizienter als bei DOM. JDOM stellt eine echte Alternative zu DOM dar. Eine Zusammenarbeit von JDOM und SAX ist auch möglich, weil JDOM in der Lage ist, als Ausgabe SAX-Ereignisse auszulösen. Diese können mit SAX-basierten Tools weiterverarbeitet werden. So lässt sich JDOM auch sehr gut in Umgebungen einsetzen, in denen weitere Tools zur Verarbeitung von XML genutzt werden.

8.7.1 JDOM beziehen

Die Projekt-Homepage *http://www.jdom.org/* bietet Download, Dokumentation und Mailinglisten. Das ZIP-Archiv *http://www.jdom.org/dist/binary/jdom-2.0.5.zip* enthält das Java-Archiv *jdom-2.0.5.jar*, das wir dem Klassenpfad hinzufügen. Die API-Dokumentation liegt online unter *http://jdom.org/docs/apidocs/index.html*. JDOM ist freie Software, die auf der Apache-Lizenz be-

ruht. Das heißt, dass JDOM auch in kommerziellen Produkten eingesetzt werden kann, ohne dass diese automatisch Open Source sein müssen.

JDOM-Version und JDOM-Alternativen

JDOM ist vor Java 5 und dem Einzug von Generics entwickelt worden, und es dauerte lange, bis JDOM 2 auf Java 5 umzog. dom4j dagegen nutzte schon recht früh Generics. XOM nutzt nach außen keine Typen der Collection-API, denn spezielle XOM-Klassen wie `Nodes` und `Elements` sind Container und liefern typisierte Objekte. Leider implementieren sie keine neuen Java-Collection-API-Klassen wie `Iterable`, und die Weiterentwicklung ist fraglich. Wenn wir im Folgenden von JDOM sprechen, meinen wir immer die aktuelle Version JDOM 2.

8.7.2 Paketübersicht *

JDOM besteht aus 15 Paketen und Unterpaketen mit den Klassen zur Repräsentation des Dokuments, zum Einlesen und Ausgeben, zur Transformation und für XPath-Anfragen. Die wichtigsten davon stellen die nächsten Abschnitte vor.

Das Paket org.jdom2

Das Paket `org.jdom2` (ab JDOM 2, vorher `org.jdom`) fasst alle Klassen zusammen, um ein XML-Dokument im Speicher zu repräsentieren. Dazu gehören zum Beispiel die Klassen `Attribute`, `Comment`, `CDATA`, `DocType`, `Document`, `Element`, `Entity` und `ProcessingInstruction`. Ein Dokument-Objekt hat ein Wurzelelement, eventuell Kommentare, einen `DocType` und eine `ProcessingInstruction`. `Content` ist die abstrakte Basisklasse und Oberklasse von `Comment`, `DocType`, `Element`, `EntityRef`, `ProcessingInstruction` und `Text`. Die Schnittstelle `Parent` implementieren alle Klassen, die `Content` haben können. Viele Schnittstellen gibt es in JDOM nicht. Andere XML-APIs verfolgen bei dieser Frage andere Ansätze; *domj4* definiert zentrale Elemente als Schnittstellen, und die pure DOM-API beschreibt alles über Schnittstellen – konkrete Objekte kommen nur aus Fabriken, und die Implementierung ist unsichtbar.

Die Pakete org.jdom2.output und org.jdom2.input

In den beiden Paketen `org.jdom2.output` und `org.jdom2.input` liegen die Klassen, die XML-Dateien lesen und schreiben können. `XMLOutputter` übernimmt die interne Repräsentation und erzeugt eine XML-Ausgabe in einen `PrintWriter`. Daneben werden die unterschiedlichen Verarbeitungsstrategien DOM und SAX durch die Ausgabeklassen `SAXOutputter` und `DOMOutputter` berücksichtigt. `SAXOutputter` nimmt einen JDOM-Baum und erzeugt benutzerdefinierte SAX2-Ereignisse. Der `SAXOutputter` ist eine sehr einfache Klasse und bietet lediglich eine `output(Document)`-Methode an. Mit `DOMOutputter` wird aus dem internen Baum ein DOM-Baum erstellt.

Ein Builder nimmt XML-Daten in verschiedenen Formaten entgegen und erzeugt daraus ein JDOM-`Document`-Objekt. Das ist bei JDOM der wirkliche Verdienst, dass unabhängig von der Eingabeverarbeitung ein API-Set zur Verfügung steht. Die verschiedenen DOM-Implementierungen unterscheiden sich an manchen Stellen. Die Schnittstelle `Builder` wird von allen einlesenden Klassen implementiert. Im `Input`-Paket befinden sich dafür die Klassen `DOMBuilder`, die einen JDOM-Baum mit DOM erzeugt, und `SAXBuilder`, die dafür SAX verwendet. Damit kann das Dokument aus einer Datei, einem Stream oder einer URL erzeugt werden. Nach dem Einlesen sind die Daten vom konkreten Parser des Herstellers unabhängig und können weiterverarbeitet werden. `SAXBuilder` ist schneller und speicherschonender. Ein `DOMBuilder` wird meistens nur dann benutzt, wenn ein DOM-Baum weiterverarbeitet werden soll.

Das Paket org.jdom2.transform

Mit dem Paket `org.jdom2.transform` wird das JAXP-TraX-Modell in JDOM integriert. Dies ermöglicht es JDOM, XSLT-Transformationen von XML-Dokumenten zu unterstützen. Das Paket enthält die beiden Klassen `JDOMResult` und `JDOMSource`. Die Klasse `JDOMSource` ist eine Wrapper-Klasse, die ein JDOM-Dokument als Parameter nimmt und diesen als Eingabe für das JAXP-TraX-Modell bereitstellt. Die Klasse `JDOMResult` enthält das Ergebnis der Transformation als JDOM-Dokument. Die beiden Klassen haben nur wenige Methoden, und in der API sind Beispiele für die Benutzung dieser Klassen angegeben.

Das Paket org.jdom2.xpath

Im Paket `org.jdom2.xpath` befindet sich nur eine Utility-Klasse `XPath`. Diese Klasse bildet die Basis für die Verwendung der Abfragesprache XPath mit JDOM. Eine kurze Einführung in XPath sowie Beispiele für den Einsatz in JDOM bietet Abschnitt 8.7.12, »XPath«. Neben der Implementierung, die mit JDOM geliefert wird, kann auch eine spezielle Implementierung der XPath-Methoden für JDOM eingesetzt werden. JDOM bringt keine eigene XPath-Implementierung mit, sondern basiert auf der Open-Source-Implementierung *Jaxen* (*http://jaxen.codehaus.org/*).

8.7.3 Die Document-Klasse

Dokumente werden bei JDOM über die Klasse `Document` verwaltet. Ein Dokument besteht aus einem `DocType`, einer `ProcessingInstruction`, einem Wurzelelement und Kommentaren. Die Klasse `Document` gibt es auch in der Standardschnittstelle für das DOM. Falls sowohl JDOM als auch DOM verwendet werden, muss für die Klasse `Document` der voll qualifizierte Klassenname mit vollständiger Angabe der Pakete verwendet werden, weil sonst nicht klar ist, welche `Document`-Klasse verwendet wird.

Ein JDOM-Document im Speicher erstellen

Um ein `Document`-Objekt zu erzeugen, bietet die Klasse drei Konstruktoren an. Über einen Standard-Konstruktor erzeugen wir ein leeres Dokument. Dieses können wir später bearbeiten, indem wir zum Beispiel Elemente (Objekte vom Typ `Element`), Entitäten oder Kommentare einfügen. Ein neues Dokument mit einem Element erhalten wir über einen Konstruktor, zu dem wir ein Wurzelelement angeben. Jedes XML-Dokument hat ein Wurzelelement.

Beispiel

Die folgende Zeile erzeugt ein JDOM-Dokument mit einem Wurzelelement:

Listing 8.30 com/tutego/insel/xml/jdom/CreateRoot.java, main()

```
Document doc = new Document( new Element("party") );
```

In XML formatiert, könnte das so aussehen:

```
<party>
</party>
```

8.7.4 Eingaben aus der Datei lesen

Ein zweiter Weg, um ein JDOM-Dokument anzulegen, führt über einen Eingabestrom oder einen Dateinamen. Dafür benötigen wir einen Builder, zum Beispiel den `SAXBuilder` (den wir bevorzugen wollen).

Beispiel

Lies die Datei *party.xml* ein:

Listing 8.31 com/tutego/insel/xml/jdom/ReadXmlFile.java, main() Teil 1

```
try ( Reader in = Files.newBufferedReader( Paths.get( "party.xml" ),
                                           StandardCharsets.UTF_8 ) ) {
  Document doc = new SAXBuilder().build( in );
  …
}
```

Die mögliche Ausnahme `JDOMException` bei `build(…)` muss die Anwendung abfangen. Kürzer ist das Programm bei XML-Dateien im Standard-Dateisystem, doch die Kodierung ist hier nicht explizit.

```
Document doc = new SAXBuilder().build( "party.xml" );
```

Bei dieser Abkürzung muss die Anwendung `IOException` und `JDOMException` behandeln.

Die Klasse `Document` bietet selbst keine Lesemethoden. Es sind immer die Builder, die `Document`-Objekte liefern. Es ist ebenso möglich, ein JDOM-Dokument mithilfe des DOM-Parsers über `DOMBuilder` zu erzeugen. Neben den Standard-Konstruktoren bei `SAXBuilder` und `DOMBuilder` lässt sich unter anderem ein `boolean`-Wert angeben, der die Validierung auf wohldefinierten XML-Code einschaltet.

Tipp

Wenn ein DOM-Baum nicht schon vorliegt, ist es sinnvoll, ein JDOM-Dokument stets mit dem SAX-Parser zu erzeugen. Das schont die Ressourcen und geht viel schneller, weil keine spezielle Datenstruktur für den DOM-Baum erzeugt werden muss. Das Ergebnis ist in beiden Fällen ein JDOM-Dokument, das die XML-Datei in einer baumähnlichen Struktur abbildet.

```
class org.jdom2.input.SAXBuilder
implements SAXEngine
```

- `SAXBuilder()`
  Baut einen XML-Leser auf Basis von SAX auf. Es wird nicht validiert.
- `SAXBuilder(boolean validate)`
  Baut einen validierenden `SAXBuilder` auf.
- `Document build(File file)`
- `Document build(InputSource in)`
- `Document build(InputStream in)`
- `Document build(InputStream in, String systemId)`
- `Document build(Reader characterStream)`
- `Document build(Reader characterStream, String systemId)`
- `Document build(String systemId)`
- `Document build(URL url)`
  Baut ein JDOM-Dokument aus der gegebenen Quelle auf. Im Fall des `String`-Arguments handelt es sich um einen URI-Namen und nicht um ein XML-Dokument im String.

8.7.5 Das Dokument im XML-Format ausgeben

Mit einem `XMLOutputter` lässt sich der interne JDOM-Baum als XML-Datenstrom in einen `OutputStream` oder `Writer` schieben.

Beispiel

Gib das JDOM-Dokument auf der Konsole aus:

Listing 8.32 com/tutego/insel/xml/jdom/ReadXmlFile.java, main() Teil 2

```
XMLOutputter out = new XMLOutputter();
out.output( doc, System.out );
```

Die Standardparametrisierung des Formatierers schreibt die XML-Daten mit schönen Einrückungen. Jeder Eintrag kommt in eine einzelne Zeile. Weitere Anpassungen der Formatierung

übernimmt ein `org.jdom2.output.Format`-Objekt. Einige statische Methoden bereiten `Format`-Objekte mit unterschiedlichen Belegungen vor, so `getPrettyFormat()` für hübsch eingerückte Ausgaben und `getCompactFormat()` mit so genannter Leerraum-Normalisierung, wie es die API-Dokumentation nennt, und `getRawFormat()`.

```
XMLOutputter out = new XMLOutputter( Format.getPrettyFormat() );
out.output( doc, System.out );
```

Unterschiedliche `setXXX(…)`-Methoden auf dem `XMLOutputter`-Objekt ermöglichen eine weitere individuelle Anpassung der `Format`-Objekte. Soll das Ergebnis als String vorliegen, kann `outputString(…)` verwendet werden, das ein `String`-Objekt liefert.

Beispiel

Konvertiere einen String in XML:

```
String s = new XMLOutputter().outputString( new Text("<Hallo\nWelt>") );
```

8.7.6 Der Dokumenttyp *

Ein XML-Dokument beschreibt in seinem Dokumenttyp den Typ der Datei und besitzt oft einen Verweis auf die beschreibende DTD.

Beispiel

Ein gültiger Dokumenttyp für XHTML-Dateien hat folgendes Format:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Bearbeiten wir dies über JDOM, so liefert die Methode `getDocType()` vom Dokument-Objekt ein `DocType`-Objekt, das wir nach den IDs fragen können. Über `setDocType(DocType)` kann der veränderte Dokumenttyp neu zugewiesen werden.

```
class org.jdom2.Document
implements Parent
```

- `DocType getDocType()`
  Liefert das zugehörige `DocType`-Objekt oder `null`, wenn keines existiert.
- `Document setDocType(DocType docType)`
  Setzt ein neues `DocType`-Objekt für das Dokument.

Beispiel

Wir erfragen vom Dokument den Elementnamen, die öffentliche ID und die System-ID:

```
DocType docType = doc.getDocType();
System.out.println( "Element: "   + docType.getElementName() );
System.out.println( "Public ID: " + docType.getPublicID() );
System.out.println( "System ID: " + docType.getSystemID() );
```

Zu den Methoden `getPublicID()` und `getSystemID()` gibt es entsprechende Setze-Methoden, nicht aber für den Elementnamen; dieser kann nachträglich nicht mehr modifiziert werden. Wir müssen dann ein neues `DocType`-Objekt anlegen. Es gibt mehrere Varianten von Konstruktoren, mit denen gesteuert werden kann, welche Einträge gesetzt werden.

Beispiel

Wir legen ein neues `DocType`-Objekt an und weisen es einem Dokument `doc` zu:

```
DocType doctype = new DocType( "html", "-//W3C...", "http://..." );
doc.setDocType( doctype );
```

8.7.7 Elemente

Jedes Dokument besteht aus einem Wurzelelement. Wir haben schon gesehen, dass dieses durch die allgemeine Klasse `Element` abgebildet wird. Mit dem Wurzelelement gelingt der Zugriff auf die anderen Elemente des Dokumentenbaums.

Wurzelelement

Die folgenden Beispieldateien verwenden die XML-Datei *party.xml*, um die Methoden von JDOM vorzustellen. Durch das Erzeugen eines leeren JDOM-Dokuments und die Methoden zur Erstellung von Elementen und Attributen kann JDOM den Dateiinhalt auch leicht aufbauen:

Listing 8.33 party.xml

```
<party datum="31.12.2012">
    <gast name="Albert Angsthase">
        <getraenk>Wein</getraenk>
        <getraenk>Bier</getraenk>
        <zustand ledig="true" nuechtern="false"/>
    </gast>
    <gast name="Martina Mutig">
        <getraenk>Apfelsaft</getraenk>
        <zustand ledig="true" nuechtern="true"/>
    </gast>
```

```
    <gast name="Zacharias Zottelig"></gast>
</party>
```

Um an das Wurzelelement `<party>` zu gelangen und von dort aus weitere Elemente oder Attribute auslesen zu können, erzeugen wir zunächst ein JDOM-Dokument aus der Datei *party.xml* und nutzen zum Zugriff `getRootElement()`.

Beispiel

Lies die Datei *party.xml*, und erfrage das Wurzelelement:

Listing 8.34 com/tutego/insel/xml/jdom/RootElement.java, main()

```
Document doc = new SAXBuilder().build( "party.xml" );
Element party = doc.getRootElement();
```

```
class org.jdom2.Document
implements Parent
```

- `Element getRootElement()`
  Gibt das `Root`-Element zurück oder `null`, falls kein `Root`-Element vorhanden ist.

Durch die oben gezeigten Anweisungen wird aus der XML-Datei *party.xml* eine JDOM-Datenstruktur im Speicher erzeugt. Um mit dem Inhalt der XML-Datei arbeiten zu können, ist der Zugriff auf die einzelnen Elemente notwendig. Durch die Methode `getRootElement()` wird das Wurzelelement der XML-Datei zurückgegeben. Dieses Element ist der Ausgangspunkt für die weitere Verarbeitung der Datei.

Zugriff auf Elemente

Um ein bestimmtes Element zu erhalten, gibt es die Methode `getChild(String name)`. Mit dieser Methode wird das nächste Unterelement des Elements zurückgegeben, das diesen Namen trägt.

Beispiel

Wenn wir den ersten Gast auf der Party haben möchten, schreiben wir:

Listing 8.35 com/tutego/insel/xml/jdom/AlbertTheFirst.java, main()

```
Element party = doc.getRootElement();
Element albert = party.getChild( "gast" );
```

Wenn wir wissen wollen, was Albert trinkt, schreiben wir:

```
Element albertGetraenk = albert.getChild( "getraenk" );
```

Durch eine Kaskadierung ist es möglich, über das Wurzelelement auf das Getränk des ersten Gastes zuzugreifen:

```
Element albertGetraenk = party.getChild( "gast" ).getChild( "getraenk" );
```


Eine Liste mit allen Elementen liefert die Methode `getChildren()`. Sie gibt eine mit `Element` generisch deklarierte `java.util.List` mit allen Elementen dieses Namens zurück.

Beispiel

Falls wir eine Gästeliste der Party haben wollen, schreiben wir:

```
List<Element> gaeste = party.getChildren( "gast" );
```

Diese Liste enthält alle Elemente der Form `<gast ...> ... </gast>`, die direkt unter dem Element `<party>` liegen.

```
class org.jdom2.Element
extends Content
implements Parent
```

- `Element getChild(String name)`
  Rückgabe des ersten untergeordneten Elements mit dem lokalen Namen `name`, das keinem Namensraum zugeordnet ist.
- `Element getChild(String name, Namespace ns)`
  Rückgabe des ersten untergeordneten Elements mit dem lokalen Namen `name`, das dem Namensraum `ns` zugeordnet ist.
- `List<Element> getChildren()`
  Rückgabe einer Liste der Elemente, die diesem Element direkt untergeordnet sind. Falls keine Elemente existieren, wird eine leere Liste zurückgegeben. Änderungen an der Liste spiegeln sich auch in der JDOM-Datenstruktur wider.
- `List<Element> getChildren(String name)`
  Rückgabe einer Liste der Elemente mit dem Namen `name`, die diesem Element direkt untergeordnet sind. Falls keine Elemente existieren, wird eine leere Liste zurückgegeben. Änderungen an der Liste spiegeln sich auch in der JDOM-Datenstruktur wider.
- `List getChildren(String name, Namespace ns)`
  Rückgabe einer Liste der Elemente mit dem Namen `name`, die diesem Namensraum zugeordnet und diesem Element direkt untergeordnet sind. Falls keine Elemente existieren, wird eine leere Liste zurückgegeben. Änderungen an der Liste spiegeln sich auch in der JDOM-Datenstruktur wider.
- `boolean hasChildren()`
  Rückgabe eines `boolean`-Werts, der ausdrückt, ob Elemente untergeordnet sind oder nicht.
- `boolean isRootElement()`
  Gibt einen Wahrheitswert zurück, der ausdrückt, ob das Element die Wurzel der JDOM-Datenstruktur ist.

8.7.8 Zugriff auf Elementinhalte

Von Beginn eines Elements bis zu dessen Ende treffen wir auf drei unterschiedliche Informationen:

- Es können weitere Elemente folgen. Im oberen Beispiel folgt in `<gast>` noch ein Element `<getraenk>`.
- Das Element enthält Text (wie das Element `<getraenk>`).
- Zusätzlich kann ein Element auch Attribute beinhalten. Dies haben wir auch beim Element `<gast>` gesehen, das als Attribut den Namen des Gastes enthält. Der Inhalt von Attributen ist immer Text.

Für diese Aufgaben bietet die `Element`-Klasse unterschiedliche Anfrage- und Setze-Methoden. Wir wollen mit dem Einfachsten, dem Zugriff auf den Textinhalt eines Elements, beginnen.

Elementinhalte auslesen und setzen

Betrachten wir das Element, dessen Inhalt wir auslesen wollen, so nutzen wir dazu die Methode `getText()`:

```
<getraenk>Wein</getraenk>
```

Sie liefert einen String, sofern eine String-Repräsentation des Inhalts erlaubt ist. Falls das Element keinen Text oder nur Unterelemente besitzt, ist der Rückgabewert ein Leer-String.

Beispiel

Um an das erste Getränk von Albert zu kommen, schreiben wir:

Listing 8.36 com/tutego/insel/xml/jdom/AlbertsDrink.java, Ausschnitt main()

```
Element party = doc.getRootElement();
Element albertGetraenk = party.getChild( "gast" ).getChild( "getraenk" );
String getraenk = albertGetraenk.getText();
```

```
class org.jdom2.Element
extends Content
implements Parent
```

- `String getText()`
  Rückgabe des Inhalts des Elements. Dies beinhaltet allen Weißraum und CDATA-Sektionen. Falls der Elementinhalt nicht zurückgegeben werden kann, wird der leere String zurückgegeben.
- `String getTextNormalize()`
  Verhält sich wie `getText()`. Weißraum am Anfang und am Ende des Strings wird entfernt.

  Mehrfach hintereinander gesetzter Weißraum innerhalb des Strings wird auf ein Leerzeichen normalisiert. Falls der Text nur aus Weißraum besteht, wird der leere String zurückgegeben.
- `String getTextTrim()`
  Verhält sich wie `getTextNormalize()`, doch Weißraum innerhalb des Strings bleibt erhalten.

Für die Methode `getText()` muss das Element vorliegen, dessen Inhalt gelesen werden soll. Mit der Methode `getChildText()` kann der Inhalt eines untergeordneten Elements auch direkt ermittelt werden.

Beispiel

Lies den Text des ersten untergeordneten Elements mit dem Namen `getraenk`. Das übergeordnete Element von Getränk ist `albert`:

Listing 8.37 com/tutego/insel/xml/jdom/AlbertsDrink.java, Ausschnitt main()

```
Element albert = party.getChild( "gast" );
String getraenk = albert.getChildText( "getraenk" );
```

In der Implementierung der Methode `getChildText(…)` sind die Methoden `getChild(…)` und `getText()` zusammengefasst.

```
class org.jdom2.Element
extends Content
implements Parent
```

- `String getChildText(String name)`
  Rückgabe des Inhalts des Elements mit dem Namen `name`. Falls der Inhalt kein Text ist, wird ein leerer String zurückgegeben. Falls das Element nicht existiert, wird `null` zurückgegeben.
- `String getChildText(String name, Namespace ns)`
  Verhält sich wie `getChildText(String)` im Namensraum `ns`.
- `String getChildTextTrim(String name)`
  Verhält sich wie `getChildText(String)`. Weißraum am Anfang und am Ende des Strings wird entfernt. Weißraum innerhalb des Strings bleibt erhalten.
- `String getChildTextTrim(String name, Namespace ns)`
  Verhält sich wie `getChildTextTrim(String)` im Namensraum `ns`.
- `String getName()`
  Rückgabe des lokalen Namens des Elements ohne Namensraum-Präfix.
- `Namespace getNamespace()`
  Rückgabe des Namensraums oder eines leeren Strings, falls diesem Element kein Namensraum zugeordnet ist.
- `Namespace getNamespace(String prefix)`
  Rückgabe des Namensraums des Elements mit diesem Präfix. Dies beinhaltet das Hochlaufen

in der Hierarchie des JDOM-Dokuments. Falls kein Namensraum gefunden wird, gibt diese Methode `null` zurück.

- `String getNamespacePrefix()`
  Rückgabe des Namensraum-Präfixes. Falls kein Namensraum-Präfix existiert, wird ein Leer-String zurückgegeben.
- `String getNamespaceURI()`
  Rückgabe des Namensraum-URIs, der dem Präfix dieses Elements zugeordnet ist, oder des Standardnamensraums. Falls kein URI gefunden werden kann, wird ein leerer String zurückgegeben.

8.7.9 Liste mit Unterelementen erzeugen *

Mit den oben beschriebenen Methoden war es bislang immer nur möglich, das erste untergeordnete Element mit einem bestimmten Namen zu lesen. Um gezielt nach bestimmten Elementen zu suchen, ist es notwendig, die untergeordneten Elemente in eine Liste zu übertragen. Mit der Methode `getContent()` wird eine Liste mit allen Elementen und Unterelementen erzeugt. Diese Liste enthält Referenzen der Elemente aus der JDOM-Datenstruktur.

Beispiel

Hole eine Liste aller Informationen der Party, und laufe sie mit einem Iterator ab:

Listing 8.38 com/tutego/insel/xml/jdom/PartyList.java, main()

```
List<Content> partyInfo = party.getContent();
Iterator<Content> partyIterator = partyInfo.iterator();
while ( partyIterator.hasNext() )
  System.out.println( partyIterator.next() );
```

```
class org.jdom2.Element
extends Content
implements Parent
```

- `List<Content> getContent()`
  Dies liefert den vollständigen Inhalt eines Elements mit allen Unterelementen. Die Liste kann Objekte vom Typ `String`, `Element`, `Comment`, `ProcessingInstruction` und `Entity` enthalten. Falls keine Elemente vorhanden sind, wird eine leere Liste zurückgegeben.

8.7.10 Neue Elemente einfügen und ändern

Um neue Elemente zu erzeugen, bietet die Klasse `Element` unter anderem den Konstruktor `Element(String)` an. Es wird ein Element mit dem entsprechenden Namen erzeugt.

Beispiel

Erfrage eine Liste mit allen Unterelementen von `albert`, erzeuge ein neues Element, und füge es in die Liste ein:

Listing 8.39 com/tutego/insel/xml/jdom/AlbertsWater.java, main()

```
Element party = doc.getRootElement();
Element albert = party.getChild( "gast" );
List<Content> albertInfo = albert.getContent();
Element wasser = new Element( "getraenk" );
wasser.addContent( "Wasser" );
```

Um den Wert eines Elements zu ändern, gibt es die Methoden `setText(…)` und `addContent(…)`. Die Methode `setText(…)` hat allerdings die unangenehme Eigenschaft, alle Unterelemente zu entfernen. Die Methode `addContent(…)` fügt neuen Inhalt hinzu.

Wenn der Inhalt eines Elements ausgetauscht werden soll, muss der alte entfernt und der neue mit `addContent(…)` hinzugefügt werden. Die Methode `addContent(…)` kann nicht nur Text, sondern jeden beliebigen Inhalt einfügen.

Beispiel

Albert will in Zukunft keinen Wein mehr trinken, sondern nur noch Wasser und Bier. Dazu wird zuerst das erste Unterelement gelöscht:

```
albert.removeChild( "getraenk" );
```

Ein neues Element `wasser` wird erzeugt und mit Inhalt gefüllt:

```
Element wasser = new Element( "getraenk" );
wasser.addContent( "Wasser" );
```

Das neue Element wird dem Element `albert` untergeordnet:

```
albert.addContent( wasser );
```

Werfen wir erneut einen Blick auf unsere XML-Datei, und entfernen wir das erste Element `<getraenk>`, das dem ersten Element `<gast>` untergeordnet ist:

```
<party datum="31.12.2012">
    <gast name="Albert Angsthase">
        <getraenk>Wein</getraenk>
        <getraenk>Bier</getraenk>
        <zustand ledig="true" nuechtern="false"/>
    </gast>
<party>
```

Beispiel

Die Methode `removeChild` entfernt das Element `<getraenk>`:

```
Element party = doc.getRootElement();
Element albert = party.getChild( "gast" );
```

Es werden nur die direkten Nachfolger durchsucht. Diese Methode findet das Element `<getraenk>Wein</getraenk>` nicht.

```
party.removeChild( "getraenk" );
```

Mit `removeChild(…)` wird das Element `<getraenk>Wein</getraenk>` gelöscht.

```
albert.removeChild( "getraenk" );
```

```
class org.jdom2.Element
extends Content
implements Parent
```

- `Element(String name)`
  Dieser Konstruktor erzeugt ein Element mit dem Namen `name` ohne Zuordnung zu einem Namensraum.
- `Element(String name, Namespace namespace)`
  Dieser Konstruktor erzeugt ein Element mit dem Namen `name` und dem Namensraum `namespace`.
- `Element(String name, String uri)`
  Dieser Konstruktor erzeugt ein neues Element mit dem lokalen Namen `name` und dem URI des Namensraums, die zu dem Element ohne Präfix gehören.
- `Element(String name, String prefix, String uri)`
  Dieser Konstruktor erzeugt ein neues Element mit dem lokalen Namen `name`, dem Namenspräfix `prefix` und dem URI des Namensraums.

Von diesen Konstruktoren ist in den Beispielen nur der erste benutzt worden.

- `boolean removeChild(String name)`
  Entfernt das erste gefundene Unterelement mit dem Namen `name`, das keinem Namensraum zugeordnet ist. Es werden nur die direkten Nachfolger durchsucht.
- `boolean removeChild(String name, Namespace ns)`
  Verhält sich wie `removeChild(String name)`. Der Namensraum wird bei der Auswahl des Elements berücksichtigt.
- `boolean removeChildren()`
  Entfernt alle untergeordneten Elemente.
- `boolean removeChildren(String name)`
  Entfernt alle Unterelemente mit den Namen `name`, die gefunden werden und keinem Namensraum zugeordnet sind. Es werden nur die direkten Nachfolger durchsucht.

- `boolean removeChildren(String name, Namespace ns)`
  Verhält sich wie `removeChildren(String)` im Namensraum `ns`.

Bei den folgenden Methoden wird als Rückgabewert das geänderte Element zurückgegeben:

- `Element setText(String text)`
  Setzt den Inhalt des Elements. Alle anderen Inhalte und alle Unterelemente werden gelöscht.
- `Element addContent(String text)`
  Ergänzt den Inhalt des Elements um den Text.
- `Element addContent(content child)`
  Ergänzt den Inhalt des Elements um das Element als Unterelement.
- `Element getCopy(String name)`
  Erzeugt eine Kopie des Elements mit dem neuen Namen `name`, ohne Zuordnung zu einem Namensraum.
- `Element getCopy(String name, Namespace ns)`
  Erzeugt eine Kopie des Elements mit dem neuem Namen `name` und eine Zuordnung zu dem Namensraum `ns`.
- `Document getDocument()`
  Liefert das Dokument dieses Elements oder `null`, falls das Element keinem Dokument zugeordnet ist.

8.7.11 Attributinhalte lesen und ändern

Ein Element kann auch einen Attributwert enthalten. Dies ist der Wert, der direkt in dem Tag mit angegeben ist. Betrachten wir dazu folgendes Element:

```
<gast name="Albert Angsthase">
```

Das Element hat als Attribut `name="Albert Angsthase"`. Diesen Wert liefert die Methode `getAttribute(String).getValue()` der Klasse `Element`.

Beispiel

Lies den Namen des ersten Gastes:

Listing 8.40 com/tutego/insel/xml/jdom/Wedding.java, main()

```
Element   party      = doc.getRootElement();
Element   albert     = party.getChild( "gast" );
Attribute albertAttr = albert.getAttribute( "name" );
String albertName    = albert.getAttribute( "name" ).getValue();
```

Martina möchte wissen, ob Albert noch ledig ist:

```
albert.getChild( "zustand" ).getAttribute( "ledig" ).getValue();
```

Auf ähnliche Weise lässt sich der Wert eines Attributs ändern. Dazu gibt es die Methoden `setAttribute(String)` der Klasse `Attribute` und `addAttribute(Attribute)` der Klasse `Element`.

Beispiel

Martina und Albert haben geheiratet, und Albert nimmt den Namen von Martina an:

```
albert.getAttribute( "name" ).setAttribute( "Albert Mutig" );
```

Seit der Hochzeit mit Albert trinkt Martina auch Wein. Also muss ein neues Element `wein` unter dem Element `<gast name="Martina Mutig">` eingefügt werden. Zuerst erzeugen wir ein Element der Form `<getraenk>Wein</getraenk>`:

```
Element wein = new Element( "getraenk" );
wein.addContent( "Wein" );
```

Danach suchen wir Martina in der Gästeliste und fügen das Element `<wein>` ein:

```
Iterator<Element> gaesteListe = party.getChildren( "gast" ).iterator();
while ( gaesteListe.hasNext() ) {
  Element gast = (Element) gaesteListe.next();

  if ( "Martina Mutig".equals(
         gast.getAttribute( "name" ).getValue()) )
    gast.addContent( wein );
}
```

Das Beispiel macht deutlich, wie flexibel die Methode `addContent(Inhalt)` ist. Es zeigt ebenso, wie JDOM für Java, etwa durch die Implementierung der Schnittstelle `List`, optimiert wurde.

```
class org.jdom2.Element
extends Content
implements Parent
```

- `Attribute getAttribute(String name)`
  Rückgabe des Attributs mit dem Namen `name`, das keinem Namensraum zugeordnet ist. Falls das Element kein Attribut mit dem Namen `name` hat, ist die Rückgabe `null`.
- `Attribute getAttribute(String name, Namespace ns)`
  Verhält sich wie `getAttribute(String)` in dem Namensraum `ns`.
- `List getAttributes()`
  Rückgabe einer Liste aller Attribute eines Elements oder einer leeren Liste, falls das Element keine Attribute hat.
- `String getAttributeValue(String name)`
  Rückgabe des Attributwerts mit dem Namen `name`, dem kein Namensraum zugeordnet ist. Es wird `null` zurückgegeben, falls keine Attribute dieses Namens existieren, und der leere String, falls der Wert des Attributs leer ist.

- `String getAttributeValue(String name, Namespace ns)`
  Verhält sich wie `getAttributeValue(String)` in dem Namensraum `ns`.
- `Element setAttributes(List attributes)`
  Fügt alle Attribute der Liste dem Element hinzu. Alle vorhandenen Attribute werden entfernt. Das geänderte Element wird zurückgegeben.
- `Element addAttribute(Attribute attribute)`
  Einfügen des Attributs `attribute`. Bereits vorhandene Attribute mit gleichem Namen und gleichem Namensraum werden ersetzt.
- `Element addAttribute(String name, String value)`
  Einfügen des Attributs mit dem Namen `name` und dem Wert `value`. Um Attribute mit einem Namensraum hinzuzufügen, sollte die Methode `addAttribute(Attribute attribute)` verwendet werden.

```
class org.jdom2.Attribute
implements NamespaceAware, Serializable, Cloneable
```

- `String getValue()`
  Rückgabe des Werts dieses Attributs

Die folgenden Methoden versuchen eine Umwandlung in einen primitiven Datentyp. Falls eine Umwandlung nicht möglich ist, wird eine `DataConversionException` ausgelöst.

- `getBooleanValue()`
  Gibt den Wert des Attributs als `boolean` zurück.
- `double getDoubleValue()`
  Gibt den Wert des Attributs als `double` zurück.
- `float getFloatValue()`
  Gibt den Wert des Attributs als `float` zurück.
- `int getIntValue()`
  Gibt den Wert des Attributs als `int` zurück.
- `long getLongValue()`
  Gibt den Wert des Attributs als `long` zurück.
- `String getName()`
  Gibt den lokalen Namen des Attributs zurück. Falls der Name die Form `[namespacePrefix]:[elementName]` hat, wird `[elementName]` zurückgegeben. Wenn der Name kein Namensraum-Präfix hat, wird einfach nur der Name ausgegeben.
- `Namespace getNamespace()`
  Gibt den Namensraum des Attributs zurück. Falls kein Namensraum vorhanden ist, wird das konstante Namensraum-Objekt `NO_NAMESPACE` zurückgegeben. Diese Konstante enthält ein Namensraum-Objekt mit dem leeren String als Namensraum.

- `String getNamespacePrefix()`
  Gibt das Präfix des Namensraums zurück. Falls kein Namensraum zugeordnet ist, wird ein leerer String zurückgegeben.
- `String getNamespaceURI()`
  Gibt den URI zurück, der zu dem Namensraum dieses Elements gehört. Falls kein Namensraum zugeordnet ist, wird ein leerer String zurückgegeben.
- `Element getParent()`
  Gibt das Element zurück, das dem Element dieses Attributs übergeordnet ist. Falls kein übergeordnetes Element vorhanden ist, wird `null` zurückgegeben.
- `String getQualifiedName()`
  Rückgabe des qualifizierten Namens des Attributs. Falls der Name die Form `[namespacePrefix]:[elementName]` hat, wird dies zurückgegeben. Ansonsten wird der lokale Name zurückgegeben.
- `Attribute setValue(String value)`
  Setzt den Wert dieses Attributs.

8.7.12 XPath

Der Standard *XPath* (*http://www.w3.org/TR/xpath20/*) bietet eine Syntax, um einzelne Knoten oder Knotenmengen aus einer XML-Struktur zu erhalten, so wie auch eine Notation im Dateisystem die Angabe einer Datei erlaubt. Der XPath-Standard wird vom W3C verwaltet und findet in vielen Bereichen Anwendung, etwa in XSLT.

XPath betrachtet die XML-Datenstruktur als Baum. Am Anfang dieses Baums steht die *XPath-Wurzel*, die sich wie üblich vor dem ersten Element des XML-Dokuments befindet. Innerhalb des Baums kann ein XPath-Ausdruck die einzelnen Elemente, deren Attribute und Werte, Verarbeitungsanweisungen und Kommentare selektieren. Die folgenden Beispiele stellen den Zugriff auf Elemente, Elementwerte, Attribute und Attributwerte vor.

Knoten(-mengen) selektieren

XPath bietet zwei Notationen zur Selektierung:

- Die einfachere Form ist die Dateisystem-Notation, die sich an den Regeln für das UNIX-Dateisystem orientiert.
- Daneben gibt es noch eine spezielle XPath-Notation, die etwas komplizierter ist.

In Tabelle 8.16 werden einige Sprachkonstrukte der beiden Notationen exemplarisch einander gegenübergestellt:

Beschreibung	XPath-Notation	Dateisystem-Notation
Zugriff auf den ersten Knoten namens `party`	`/child::party`	`/party`
übergeordnete Knoten verwenden	`/child::party/child::gast/➲ child::zustand/parent::node()`	`/party/gast/zustand/..`
der erste Gast unserer Party	`/child::party/child::gast[1]`	`/party/gast[1]`
alle ledigen Gäste	`/child::party/child::gast/➲ child::zustand/➲ attribute::ledig[ (child::text() = ➲ "true")]`	`/party/gast/➲ zustand[@ledig="true"]`

Tabelle 8.16 Dateisystem-Notation und XPath-Notation im Vergleich

Es gibt ebenso die Möglichkeit, auf Geschwisterknoten und den aktuellen Knoten zuzugreifen. Zudem können Knoten in Abhängigkeit zu der Position im XML-Dokument und bestimmten Werten von Elementen und Attributen abgefragt werden. Um die Beispiele einfach zu halten, wollen wir allerdings immer die Dateisystem-Notation verwenden.

XPath-Tools

Im Code Strings mit XPath-Ausdrücken für XML-Dateien zu nutzen ist nicht immer so praktisch, vor allen Dingen wenn der Ausdruck komplexer ist. Es bietet sich daher an, XML-Dokumente in ein entsprechendes Werkzeug zu laden und über eine grafische Oberfläche die XPath-Ausdrücke zu testen, bis sie »passen«. Eines der Tools ist etwa der *XPath Visualizer* von Microsoft unter *https://xpathvisualizer.codeplex.com/*.

XPath-APIs

So, wie es unterschiedliche APIs zur Repräsentation der XML-Bäume im Speicher gibt (W3C DOM, JDOM ...), gibt es auch mehrere XPath-APIs. Zwei wichtige sind:

- **DOM Level 3 XPath**: Eine programmiersprachenunabhängige API ausschließlich für Bäume nach dem offiziellen W3C-DOM-Modell. Das Paket `javax.xml.xpath` setzt diese API in Java um.
- **Jaxen**: eine Java-API, die unterschiedliche DOM-Modelle wie DOM, JDOM und dom4j zusammenbringt

XPath mit JDOM

Um XPath-Anfragen mit JDOM durchzuführen, stehen die Typen `org.jdom2.xpath.XPathFactory` (nicht `javax.xml.xpath.XPathFactory`!) und `XPathExpression` (nicht `javax.xml.xpath.XPathExpression`!) im Zentrum. Im Hintergrund arbeitet standardmäßig Jaxen. JDOM bringt das nötige

Java-Archiv für Jaxen mit. Damit die folgenden Beispiele laufen, muss aus dem Ordner *lib* des Archivs *jdom-xyz.zip* die JAR-Datei *jaxen-1.1.6.jar* in den Klassenpfad aufgenommen werden. Unter *http://jaxen.codehaus.org/releases.html* lässt sich die jeweils aktuelle Version beziehen, Updates sind aber selten.

Zu einem `XPathFactory`-Objekt führt die Fabrikmethode `XPathFactory.instance()`. Diesem Objekt wird im nächsten Schritt der eigentliche XPath-Ausdruck übergeben: `XPathFactory.instance().compile(xpath)`. Das Ergebnis ist ein `XPathExpression`-Objekt, welches wir fragen können, welche Knoten in der Ergebnismenge liegen. Der XPath-Ausdruck legt fest, was aus dem XML-Dokument gewünscht ist. Die Liste kann Elemente, Attribute oder Strings enthalten, daher sind keine Typinformationen vorhanden und Generics helfen nicht viel.

Beispiel

Gib die Namen aller Gäste aus:

Listing 8.41 com/tutego/insel/jdom/xpath/XPathDemo1.java, Ausschnitt

```
XPathExpression<Object> xpath = XPathFactory.instance().compile( "/party/gast/@name" );
List<Object> names = xpath.evaluate( doc );
for ( Object object : names ) {
  Attribute attribute = (Attribute) object;
  System.out.println( attribute.getValue() );
}
```

Da es keine Typinformationen gibt, liefert `evaluate(…)` immer nur eine Liste von unbekannten Objekten, die von uns über eine explizite Typanpassung in einen sinnvollen Typ gebracht werden müssen.

Beispiel

Selektiere mit einem XPath-Ausdruck die Getränke der Gäste, und gib sie auf den Bildschirm aus:

Listing 8.42 com/tutego/insel/jdom/xpath/XPathDemo2.java, Ausschnitt

```
XPathExpression<Object> xpath = XPathFactory.instance().compile( "/party/gast/
getraenk" );
for ( Object object : xpath.evaluate( doc ) )
  System.out.println( ((Element)object).getValue() );
```

Das Ergebnis dieser beiden Aufrufe ist immer eine Knotenmenge. Es gibt aber auch Situationen, in denen nur das erste Element der Ergebnisliste verarbeitet werden soll oder nur ein Element als Ergebnis bei einem XPath-Ausdruck möglich ist, wie zum Beispiel bei der Abfrage von Elementen mit Index-Angabe. Für diesen Fall bietet die Klasse `XPathExpression` die praktische Methode `evaluateFirst(Object context)`.

Beispiel

Gib den Namen des ersten Gastes aus:

Listing 8.43 com/tutego/insel/jdom/xpath/XPathDemo3.java, Ausschnitt

```
XPathExpression<Object> xpath = XPathFactory.instance().compile( "/party/gast[1]/
@name" );
Object firstGuest = xpath.evaluateFirst( doc );
System.out.println( ((Attribute) firstGuest).getValue() );
```

```
abstract class org.jdom2.xpath.XPathFactory
```

- `static XPathFactory newInstance()`
  Liefert eine `XPathFactory` basierend auf der Standard-XPath-Implementierung Jaxen.
- `XPathExpression<Object> compile(String expression)`
  Übersetzt einen XPath-Ausdruck. Weitere überladene Methoden von `compile(…)` erlauben Filter und Variablen.

```
interface org.jdom2.xpath.XPathExpression<T>
extends java.lang.Cloneable
```

- `List<T> evaluate(Object context)`
  Wertet den vorcompilierten XPath-Ausdruck auf dem übergebenen Inhalt aus.
- `T evaluateFirst(Object context)`
  Liefert das erste Ergebnis vom ausgewerteten XPath-Ausdruck.
- `String getExpression()`
  Liefert den XPath-Ausdruck aus `String`.
- Weitere Methoden können Variablen setzen und Diagnosen durchführen.

Nutzen von XPath-Ausdrücken

Die Möglichkeiten von XPath können als Alternative zu den Zugriffen über die Datenstrukturen von Java betrachtet werden. Es ist häufig einfacher, mit einem XPath-Ausdruck als mit einzelnen Methodenaufrufen den Pfad zu den Inhalten zu kodieren. Eine Anwendung, die dem Benutzer einen Zugriff auf die XML-Daten bietet, sollte auf jeden Fall XPath anbieten, weil dies der Standard für den Zugriff ist.

Speziell für Datenbanken, die sich auf die Speicherung von XML-Dokumenten spezialisiert haben, ist es üblich, XPath als Abfragesprache zu verwenden. Als Standard in diesem Umfeld gilt XQuery, das eine SQL-ähnliche deklarative Syntax bietet. Ebenso wird XPath im Standard XSLT verwendet, um Knoten für die Umwandlung auszuwählen. Wir stellen diesen Standard im nächsten Abschnitt kurz vor.

8.8 Transformationen mit XSLT *

XSLT ist eine XML-Applikation zur Umwandlung von XML-Dateien in andere textbasierte Dateien. Die Ausgabedatei kann ein beliebiges Format haben. Die XSLT-Datei, in der Umwandlungsregeln festgelegt werden, muss ebenfalls eine XML-Datei sein. Das bedeutet insbesondere, dass HTML-Tags in der XSLT-Datei die Regeln für XML-Elemente erfüllen müssen.

8.8.1 Templates und XPath als Kernelemente von XSLT

In der XSLT-Datei werden die Elemente der XML-Quelldatei durch Templates ausgewählt und wird die Formatierung der Ausgabe beschrieben. Es ist möglich, die Formatierung von Bedingungen abhängig zu machen, Elemente in der Ausgabe auszublenden und die Reihenfolge der Ausgabe festzulegen.

Die Auswahl der Elemente wird durch *XPath*-Ausdrücke beschrieben. XPath ist eine XML-Applikation, in der eine XML-Datei als Baumstruktur abgebildet wird. Durch eine Notation, die an die Baumstruktur von Verzeichnisbäumen angelehnt ist, können einzelne Elemente oder ganze Unterbäume ausgewählt werden.

Für unser Beispiel ist hier eine einfache XSLT-Datei angegeben, die eine XML-Ausgabe aus der Datei *party.xml* erzeugt. Dabei wird in dem ersten Template ein HTML-Rumpf erzeugt, in den die Ausgabe der anderen Templates eingebettet wird. Mit dem Element `party` wird eine Überschrift für die Ausgabedatei erzeugt. Das Element `<gast>` wird in einem Template benutzt, um für jeden Gast eine persönliche Anrede zu erzeugen. Jedem Gast wird sein Lieblingsgetränk serviert. Zum Schluss beschreibt noch jeder kurz, wie es ihm geht und ob er noch ledig ist. Hier ist die XSLT-Datei für die Umwandlung:

party.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<!-- Match auf das Root-Element des XPath-Baums -->
<!-- Ausgabe von HTML-Kopf und -Fuss -->
   <xsl:template match="/">
      <html>
         <head>
            <title>Wir machen eine Party</title>
         </head>
         <body>
<!-- An dieser Stelle wird tiefer in den XPath-Baum -->
<!-- verzweigt. Die Ausgabe der anderen Templates   -->
<!-- wird an dieser Stelle eingefuegt               -->
            <xsl:apply-templates />
         </body>
      </html>
   </xsl:template>
```



```
<!-- Fuer das Element party der XML-Datei wird eine Ueberschrift -->
<!-- fuer die Ausgabe erzeugt. Das Attribut Datum wird in der   -->
<!-- Ueberschrift ausgegeben. -->
   <xsl:template match="party">
      <h1>Partytabelle fuer den
      <xsl:value-of select="@datum" />
      </h1>
      <xsl:apply-templates />
   </xsl:template>
<!-- Fuer jeden einzelnen Gast wird eine Begruessung ausgegeben -->
   <xsl:template match="gast">
      <p>
         <h2>Hallo
         <xsl:value-of select="@name" />
         </h2>
      </p>
      <xsl:apply-templates />
   </xsl:template>
<!-- Jedem Gast wird sein Lieblingsgetraenk angeboten. -->
   <xsl:template match="getraenk">
      <p>Hier ist ein
      <xsl:value-of select="." />
      fuer dich.</p>
   </xsl:template>
<!-- Hier wird eine bedingte Ausgabe erzeugt. Jeder Gast -->
<!-- zeigt seinen Zustand und sagt, ob er noch ledig ist. -->
   <xsl:template match="zustand">
      <xsl:if test="@nuechtern='true'">
         <h3>Ich bin noch nuechtern!</h3>
      </xsl:if>
      <xsl:if test="@ledig='true'">
         <h3>Ich bin noch zu haben!</h3>
      </xsl:if>
      <hr />
   </xsl:template>
</xsl:stylesheet>
```

Das Ergebnis der Umwandlung ist folgende HTML-Datei:

```
<html>
  <head>
    <title>Wir machen eine Party</title>
  </head>
  <body>
    <h1>Partytabelle fuer den 31.12.12</h1>
```

```
    <p><h2>Hallo Albert Angsthase</h2></p>
    <p>Hier ist ein Wein fuer dich.</p>
    <p>Hier ist ein Bier fuer dich.</p>
    <h3>Ich bin noch zu haben!</h3>
    <hr>
    <p><h2>Hallo Martina Mutig</h2></p>
    <p>Hier ist ein Apfelsaft fuer dich.</p>
    <h3>Ich bin noch nuechtern!</h3>
    <h3>Ich bin noch zu haben!</h3>
    <hr>
    <p><h2>Hallo Zacharias Zottelig</h2></p>
  </body>
</html>
```

Die Ausgabe des Parsers ist nicht so schön formatiert, aber das ist für die Ausgabe in HTML nicht relevant. Diese Datei wurde nachträglich formatiert, damit die Ausgabe besser lesbar ist. Trotz der Definition des Zeichensatzes im Kopf der XSLT-Datei sind Umlaute immer noch ein Problem.

8.8.2 Umwandlung von XML-Dateien mit JDOM und JAXP

Damit die Umwandlung mit JDOM möglich ist, wird das JDOM-Dokument von einem Wrapper-Objekt aufgenommen und mit einem XSLT-Parser von JAXP umgewandelt. Das Ergebnis ist ein Wrapper-Objekt und kann in eine JDOM-Datenstruktur umgewandelt werden:

Listing 8.44 com/tutego/insel/xml/xslt/XslTransformer.java, main()

```
Document   doc        = new SAXBuilder().build( "party.xml" );
Source     xmlFile    = new JDOMSource( doc );
JDOMResult htmlResult = new JDOMResult();
Transformer transformer =
   TransformerFactory.newInstance().newTransformer(
      new StreamSource("party.xsl") );
transformer.transform( xmlFile, htmlResult );
XMLOutputter xmlOutputter = new XMLOutputter();
xmlOutputter.output( doc, System.out );
xmlOutputter.output( htmlResult.getDocument(), System.out );
```

Das JDOM-Dokument `doc` wird vom Objekt `xmlFile` vom Typ `JDOMSource` ummantelt, das ein `javax.xml.transform.Source` ist. Das Ergebnis der Transformation nimmt ein `JDOMResult`-Objekt entgegen, das vom Typ `javax.xml.transform.Result` ist.

Eine Fabrikmethode der `TransformerFactory` erzeugt ein Objekt der Klasse `Transformer`. Dies ist ein Objekt aus dem JAXP-Paket, und es übernimmt die Umwandlung von XML-Dateien mithilfe einer XSLT-Datei. Für die Ein- und Ausgabe können Streams, SAX-Eigenschaften oder eine DOM-

Datenstruktur verwendet werden. In diesem Beispiel wird die Datei als Stream an den Transformer übergeben. Mit dem `Transformer`-Objekt und einem `Source`- und `Result`-Objekt aus JAXP stößt `transform(…)` die Übersetzung an.

Das Ergebnis der Umwandlung steht in unserem `JDOMResult`-Objekt, und die Methode `getDocument()` wandelt es wieder in eine JDOM-Datenstruktur um.

Dieses Beispiel zeigt das Zusammenspiel von JDOM und JAXP. JDOM ist kein Ersatz für JAXP, sondern bietet eine komfortable Möglichkeit, XML-Dateien mit einer gewohnten Java-API zu verarbeiten. Dabei können Elemente aus JAXP, die nicht in JDOM implementiert sind, genutzt werden, ohne auf die Vorteile von JDOM zu verzichten.

8.9 XML-Schema-Validierung *

XML ist ein sehr freies Format, und Dokumente können leicht XML-Elemente enthalten, die so eigentlich vom Designer nicht vorgesehen waren. Um sicherzustellen, dass XML-Dokumente nicht zu »frei« sind, sondern sich an bestimmte Regeln halten, gibt es unterschiedliche Korrektheitsbeschreibungen. Die populärste ist *XML Schema* (siehe auch Abschnitt 8.2.3, »Schema – die moderne Alternative zu DTD«), die das ältere DTD ersetzt hat. Eine XML-Schema-Datei hat ebenfalls das XML-Format (anders als DTD) und wird einem Parser zusammen mit der XML-Datei gegeben. Der liest ein XML-Dokument ein und prüft, ob die Regeln eingehalten werden.

8.9.1 SchemaFactory und Schema

Java unterstützt die Schema-Validierung, die jedoch standardmäßig ausgeschaltet ist. Ein Grund ist, dass die Validierung die Rechenzeit und den Speicherbedarf erhöht. Um die Validierung zu aktivieren, muss zunächst ein `Schema`-Objekt aufgebaut werden. Bei JAXB haben wir die Validierung schon genutzt und Folgendes geschrieben:

```
SchemaFactory sf = SchemaFactory.newInstance(
  XMLConstants.W3C_XML_SCHEMA_NS_URI );
Schema schema = sf.newSchema(
  ValidatingRoomUnmarshaller.class.getResource( "/schema1.xsd" ) );
```

Das Objekt vom Typ `Schema` repräsentiert die Schema-Beschreibung.

8.9.2 Validator

Im Fall von JAXB wurde das `Schema`-Objekt direkt an JAXB festgemacht und zur Validierung genutzt. Unabhängig davon bietet das `Schema`-Objekt die zentrale Methode `newValidator()`, die einen `Validator` liefert, der letztlich die Validierung durchführt.

```
Schema schema = ...
Validator validator = schema.newValidator();
```

abstract class javax.xml.validation.**SchemaFactory**

- `final SchemaFactory newInstance(String schemaLanguage)`
  Liefert eine `SchemaFactory`, die die angegebene Schema-Sprache unterstützt. Bei der XML-Schema-Validierung wird der String immer `XMLConstants.W3C_XML_SCHEMA_NS_URI` sein, also `"http://www.w3.org/2001/XMLSchema"`. Es gibt zwar auch die Konstante `RELAXNG_NS_URI` für *RELAX NG Schema*-Validierung, doch JAXP unterstützt dies nicht.
- `Schema newSchema(File schema) throws SAXException`
  Parst die Datei und baut das `Schema`-Objekt aus. Gibt es beim Parsen Fehler, folgt die `SAXException`.
- `Schema newSchema(URL schema) throws SAXException`
  Holt sich die Schema-Datei von der URL, parst sie und baut das `Schema`-Objekt auf. Gibt es beim Parsen Fehler, folgt die `SAXException`.

abstract class javax.xml.validation.**Schema**

- `abstract Validator newValidator()`
  Liefert einen `Validator` für das Schema.

8.9.3 Validierung unterschiedlicher Datenquellen durchführen

Dieser `Validator` bietet die Methode `validate(…)`, und ihr wird entweder eine `DOMSource`, `JAXBSource`, `SAXSource`, `StAXSource` oder `StreamSource` – also ganz allgemein gesprochen ein Objekt vom Typ `javax.xml.transform.Source` – übergeben. An dem Paket lässt sich ablesen, dass `Source` ein Typ aus dem XSLT-Paket ist.

Wie ist nun der Weg, um etwa einen DOM-Baum zu validieren? Der DOM-Baum muss als `Source`-Objekt verpackt und dann der `validate(…)`-Methode übergeben werden. Etwa so:

```
Document document = ...
validator.validate( new DOMSource(document) );
```

Gibt es Fehler, gibt es eine `SAXException`, die eine geprüfte Ausnahme ist, also behandelt werden muss. Es lässt sich mit `setErrorHandler(…)` ein spezieller `ErrorHandler` setzen, damit Fehler dort aufgefangen und dokumentiert werden können.

abstract class javax.xml.validation.**Validator**

- `void validate(Source source) throws SAXException, IOException`
  Führt die Validierung der Eingabe durch.
- `abstract void setErrorHandler(ErrorHandler errorHandler)`
  Setzt ein Objekt zum Auffangen der Fehler.

8.10 Zum Weiterlesen

Als XML sich etablierte, bildete es mit Java ein gutes Gespann. Einer der Gründe lag in Unicode: XML ermöglicht Dokumente mit beliebigen Zeichenkodierungen, die in Java abgebildet werden konnten. Mittlerweile ist diese Abbildung nicht mehr so einfach, da in XML schnell eine Kodierung mit 32 Bit ausgewählt werden kann, die in Java nur Surrogate abbildet – nun macht die Verarbeitung nicht mehr richtig Spaß.

Das online unter *http://tutego.de/go/xmlbook* frei verfügbare Buch »Processing XML with Java« von Elliotte Rusty Harold gibt einen guten Überblick über die Funktionen von JDOM und die Verarbeitung von XML mit Java. Genauere Informationen finden sich auf der Website von JDOM (*http://www.jdom.org/*). Zur JAXB gibt *http://jaxb.java.net/tutorial/* tiefere Einblicke. JAXB ist sehr nützlich, insbesondere für eigene XML-Formate; für Standardformate wie RSS-Feeds, SVG, MathML, OpenDocument oder XUL gibt es in der Regel schon Zugriffsklassen, sodass die Daten nicht aus rohen XML-Dokumenten extrahiert werden müssen – eine objektorientierte Vorgehensweise ist immer besser, als in XML-Zeichenketten direkt zu lesen und diese zu verändern. XPath als Anfragesprache für XML auch für beliebige Java-Objekte zu erweitern, hat sich das Apache-Projekt *JXPath (http://tutego.de/go/jxpath)* zum Ziel gesetzt. Es ist auf jeden Fall einen Blick wert.

XML wird im Internet mehr und mehr durch JSON abgelöst. Die Java SE bringt zur Verarbeitung nichts mit, wohl aber Java EE und auch Jersey, und zwar die REST-API, die in Kapitel 15, »RESTful und SOAP-Web-Services«, vorgestellt wird.

Auf einen Blick

Inhalt

12 JavaFX 1031

13 Netzwerkprogrammierung 1077

21 Datenbankmanagement mit JDBC 1337

22 Java Native Interface (JNI) 1383

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Christian Ullenboom, Diplom-Informatiker, ist Sun-zertifizierter Java-Programmierer und seit 1997 Trainer und Berater für Java-Technologien sowie objektorientierte Analyse und Design. Seit Jahren teilt er sein Wissen mit den unzähligen Besuchern seiner Website, wo er Fragen beantwortet, Inhalte bereitstellt und diskutiert. Seine Sympathie gilt Java Performance Tuning und den sinnlichen Freuden des Lebens.

Christian Ullenboom

Java SE 8 Standard-Bibliothek – Das Handbuch für Java-Entwickler

1.448 Seiten, gebunden, 2. Auflage 2014
49,90 Euro, ISBN 978-3-8362-2874-9

www.galileo-press.de/3607