



Uwe  
Rozanski

# Enterprise JavaBeans 3.1

**Einstieg, Umstieg, Praxis und Referenz**



inklusive CD-ROM

# Grundlagen Java EE

## 1.1 Überblick

Java EE steht für Java Enterprise Edition und ist ein Standard für die Programmierung verteilter Anwendungen. Verteilt ist eine Anwendung immer dann, wenn nicht alle Komponenten auf demselben Computer, oder um noch genauer zu sein, nicht in derselben Java VM (virtual machine) laufen. Die Enterprise Edition baut auf der Standard Edition (Java SE) auf und erweitert diese um eine Reihe von Technologien. Schwerpunkt dieses Buches ist die Betrachtung der Enterprise JavaBeans (EJB) in der Version 3.1 und der Java Persistence API. Wer schon Erfahrungen in der EJB-Programmierung hat, kann dieses Kapitel überspringen. Eventuell von Interesse könnten jedoch die Ausführungen zu Annotations und Generics am Ende dieses Kapitels sein.

An dieser Stelle wird eine Einführung in die EJB-Programmierung gegeben, ohne sich in Details zu verlieren. Alle offenen Fragen werden dann in den folgenden Kapiteln geklärt.

### 1.1.1 Architektur

Als einfaches Beispiel für eine verteilte Anwendung soll ein kleines Programm dienen, das überprüft, ob eine eingegebene Bankleitzahl gültig, also vergeben ist. Dazu benötigt man zunächst eine Datenbanktabelle, in der alle gültigen Bankleitzahlen gespeichert sind. In Abbildung 1.1 ist der Aufbau einer solchen Tabelle dargestellt, die später im Übungsteil des Kapitels über den Entity Manager (Kapitel 10) mit echten Inhalten gefüllt wird.

BANK
DATENSATZNR PRIMARY KEY, BLZ INTEGER, BANK CHAR(27), PRUEFZIFFER CHAR(2)

Abb. 1.1: Aufbau der Datenbanktabelle BANK

Um den beschriebenen Service anzubieten, genügt es, ein einfaches Java-Programm zu schreiben, das zunächst den Benutzer nach der zu überprüfenden Bankleitzahl fragt, dann eine Verbindung zur Datenbank aufbaut und nachsieht, ob es dort mindestens eine Zeile mit dieser Bankleitzahl finden kann. Wenn ja, ist sie gültig. Es würde sich aber in diesem Fall nicht um eine verteilte Anwendung handeln, da alles in einem Programm geschieht. Bei einem solch kleinen Beispiel ist das völlig unkritisch, für eine komplexe Anwendung hingegen, mit der gleichzeitig viele Personen arbeiten sollen, eine ungünstige Architektur.

### 1.1.2 Zielsetzung

Ziel von Java EE ist die Aufteilung der Anwendungslogik in einen Client- und einen Serverteil. Alle fachlichen Funktionen sollen zentral auf einem Server zur Verfügung gestellt werden, auf den alle Clients zugreifen. Dort finden auch alle Datenbankzugriffe statt. Damit die Serverkomponenten erreichbar und verwaltbar sind, werden sie unter die Kontrolle eines Applikationsservers gestellt, das ist sozusagen ein Stück Software, das den Java-EE-Standard implementiert hat. In diesem Buch wird der kostenlose Applikationsserver JBoss benutzt, der auf der beiliegenden CD enthalten ist. In dem Kapitel über die Entwicklungsumgebung (Kapitel 2) wird aber auch gezeigt, wie man sich die aktuellste Version dieses Servers besorgen kann. Für die Programmierung in der Praxis sind auch die Applikationsserver WebLogic von Oracle (ehemals BEA) und WebSphere von IBM relevant. Diese gibt es derzeit nicht kostenlos. Wenn man sich als Entwickler an den EJB-Standard hält, und das werden wir in diesem Buch machen, dann laufen die Anwendungen ohne weitere Änderungen auch auf anderen Servern.

Kommen wir zurück zur verteilten Architektur. Für unser kleines Beispiel bedeutet das, dass der Teil, der die Bankleitzahl überprüft, auf dem Server läuft. Man bezeichnet die entsprechende Java-Klasse als Bean, genauer als Session Bean. Die Oberfläche, die den Benutzer nach der Nummer fragt, ist der Client, dessen weitere Aufgabe darin besteht, eine Verbindung mit dem Applikationsserver aufzunehmen, um die dort gelagerte Bean zu verwenden. Der Beanbegriff wird bei Java an den unterschiedlichsten Stellen verwendet. Immer handelt es sich um eine Komponente, die über bestimmte Methoden verfügen muss, damit sie in einer sie umgebenden Software funktionieren. So gibt es Beans für die Programmierung grafischer Oberflächen unter Swing, Beans für die Bereitstellung der Daten für JSF-Seiten im Internet und eben auch Beans, die die fachliche Logik in einer verteilten Anwendung aufnehmen beziehungsweise für den Zugriff auf die Datenbank verantwortlich sind.

In Abbildung 1.2 sind die Zusammenhänge einmal grafisch aufbereitet. Dabei soll man davon ausgehen, dass Client und Applikationsserver auf unterschiedlichen Rechnern laufen.

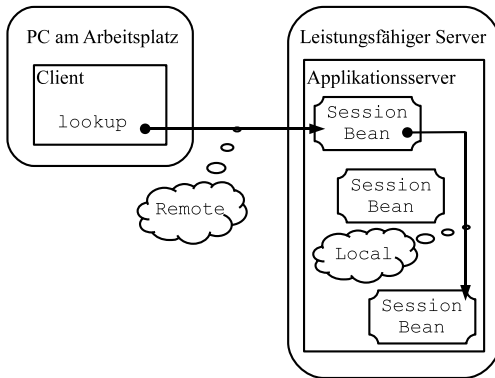


Abb. 1.2: Architektur einer EJB-Anwendung

### 1.1.3 Komponenten

Man unterscheidet verschiedene Beanarten, die in Abbildung 1.3 dargestellt sind. Unter dem Überbegriff EJB finden sich Session Beans, die die Anwendungslogik repräsentieren, Message-Driven Beans, die ebenfalls Logik beinhalten, jedoch über Nachrichten gesteuert werden, und Entity Beans, die die Daten einer Datenbanktabelle repräsentieren. Die Session Beans lassen sich nochmals in zustandsbehaftete (Stateful Session Beans) und zustandslose (Stateless Session Beans und Singleton Session Beans) aufteilen. Will man nur einen Dienst wie die Überprüfung der Bankleitzahl anbieten, wird man sich für eine zustandslose Bean entscheiden, weil serverseitig nichts weiter verwaltet werden muss. Man kann sie auch als Webservice einsetzen. Ganz anders sieht es aus, wenn ein benutzerspezifischer Warenkorb verwaltet werden soll. Das ist Aufgabe einer zustandsbehafteten, also einer Stateful Session Bean.

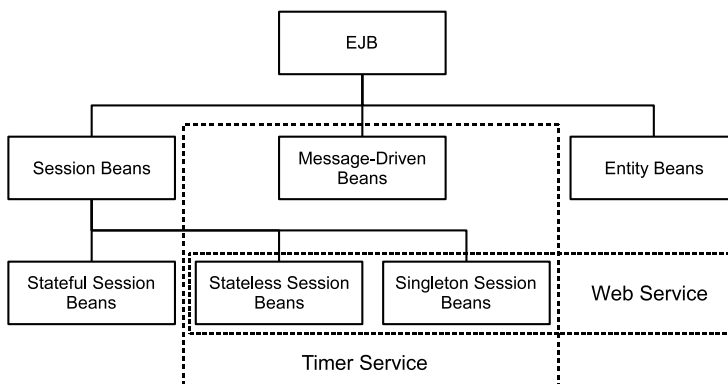


Abb. 1.3: Unterschiedliche Beanarten

## 1.2 Eine zustandslose Bean

### 1.2.1 Beanklasse

Wie erwähnt ist unser Bankleitzahlenservice in einer zustandslosen Bean bestens aufgehoben. Der Benutzer macht eine Anfrage, bekommt ein Ergebnis und damit ist der Vorgang abgeschlossen. Es ist nicht notwendig, sich auf dem Server irgendwelche Informationen über diesen Aufruf für einen nächsten zu speichern, er ist daher zustandslos. Die komplette Beanklasse ist in Listing 1.1 abgedruckt. Wie man sieht, handelt es sich dabei um eine ganz normale Java-Klasse, die mit der Annotation `@Stateless` versehen wurde. Das genügt, um aus ihr eine zustandslose Session Bean zu machen. Sie beinhaltet die Methode `blzGueltig()`, die die übergebene Bankleitzahl prüft und einen entsprechenden booleschen Wert als Ergebnis liefert.

```
package server.kap01;

import javax.ejb.Stateless;
import javax.persistence.*;

@Stateless
public class BankBean implements BankRemote {
    @PersistenceContext(unitName = "JavaEE")
    private EntityManager manager;

    public boolean blzGueltig(Integer blz) {
        Query query = manager.createNamedQuery("Bank.findByBlz");
        query.setParameter("blz", blz);
        if(query.getResultList().size() == 0)
            return false;
        else
            return true;
    }
}
```

**Listing 1.1:** Zustandslose Session Bean

Wie bereits beschrieben, muss man dafür sorgen, dass diese Klasse von einem Applikationsserver verwaltet werden kann. Dazu wird sie in den Hoheitsbereich dieses Servers gebracht, man sagt, man deployt die Klasse, man macht sie bekannt. Zu diesem Zweck verpackt man sie in ein Java-Archiv, eine ZIP-Datei mit der Endung `.jar`, und ruft das entsprechende Deploymenttool des Applikationsservers auf. Für JBoss genügt es, diese JAR-Datei in das `deploy`-Verzeichnis des Servers zu kopieren. Für andere Server ist es sogar notwendig, die JAR-Datei in

eine weitere ZIP-Datei mit der Endung `.ear` einzupacken. Man spricht dann von einem Enterprise-Archiv.

Wie man die ganze Software installiert und seine Beans deployt, ist in dem Kapitel über die Entwicklungsumgebung (Kapitel 2) beschrieben. Alles Notwendige findet sich auf der beiliegenden CD.

### 1.2.2 Remote-Interface

Jede Beanklasse bietet bestimmte Methoden an, die von einem Client aufgerufen werden können. Daneben kann es natürlich auch Methoden geben, die nicht öffentlich sein sollen. Zu diesem Zweck gehört zu jeder Session Bean mindestens ein Interface, in dem die öffentlichen Methoden aufgeführt sind. Diese Interfaces gibt es in zwei Ausprägungen, eine für entfernte Methodenaufrufe (Remote-Interface) und eine für Aufrufe aus derselben Java VM (Local-Interface). Üblicherweise implementiert eine Beanklasse mindestens eines dieser Interfaces, es kann aber auch sein, dass sie beide implementiert und dann sowohl remote als auch lokal benutzt werden kann. Das Remote-Interface der Beanklasse für die Bankleitzahlprüfung ist in Listing 1.2 wiedergegeben. Es ist ein ganz normales Java-Interface, das mit der Annotation `@Remote` versehen ist. Diese Angabe kennzeichnet es als Remote-Interface. Als einzige Methode bietet es `blzGueItig()` mit der bekannten Signatur an (erwartet Bankleitzahl und liefert boolesches Ergebnis).

```
package server.kap01;

import javax.ejb.Remote;

@Remote
public interface BankRemote {
    public boolean blzGueItig(Integer blz);
}
```

**Listing 1.2:** Remote-Interface

Dieses Interface muss sowohl dem Applikationsserver als auch dem Client bekannt sein. Für den Server packt man es dazu in dieselbe JAR-Datei, in der sich auch die Beanklasse befindet, und deployt es dadurch mit. Es muss aber auch Teil der Client-JAR-Datei sein, die man als Java-Entwickler erstellen muss, um den Client auszuliefern zu können.

Wenn der Client außerhalb des Applikationsservers gestartet wird, muss man sich ein paar Gedanken darüber machen, wie die Kommunikation zwischen dem Client und dem Server funktioniert. In der Clientanwendung hat man ein Interface, an dem man Methoden aufruft. Diese Aufrufe müssen nun irgendwie an den Server geschickt und dort tatsächlich abgearbeitet werden. Außerdem muss das

Ergebnis auf irgendeine Art und Weise in das Hoheitsgebiet des Clients zurückkommen.

Um diese Aufgabe zu lösen, wird zwischen dem Client und dem Server eine virtuelle Kommunikation aufgebaut, wie sie in Abbildung 1.4 dargestellt ist. Dazu meldet sich der Client zunächst beim Applikationsserver an und teilt diesem mit, dass er an dem Bankleitzahlservice interessiert ist. Als Antwort bekommt er eine Instanz einer Java-Klasse geliefert, die das Remote-Interface `BankRemote` implementiert hat, also über die gewünschte Methode verfügt. In Wirklichkeit hat diese Klasse aber keine Ahnung, wie man eine Bankleitzahl prüft, sondern lediglich, wie man diesen Methodenaufruf an den Server weiterleiten kann, um dann auf das Ergebnis zu warten, das anschließend als das eigene ausgegeben wird. Eine solche Java-Klasse wird *Stub* genannt und entbindet den Anwendungsentwickler von jeglichem Aufwand für das Betreiben der Kommunikation.

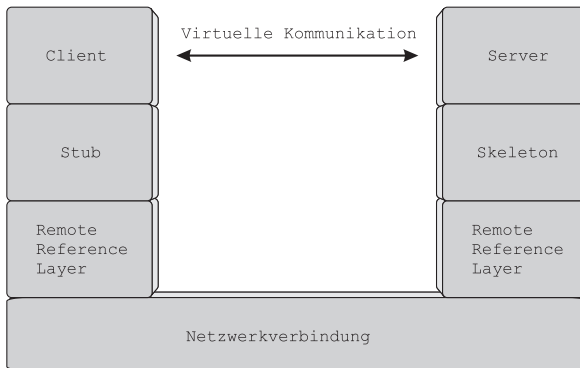


Abb. 1.4: Verbindung zwischen Client und Server

Die Stub-Klasse schickt also die Anfrage an den Server, wo ihr exaktes Gegenstück, eine Skeleton-Klasse auf sie hört. Diese nimmt den Aufruf mit seinen Parametern entgegen und ruft nun endlich die eigentliche Methode der Beanklasse auf, die die Bankleitzahl prüft. Das Ergebnis wird zunächst von der Beanklasse an die Skeleton-Klasse und von dieser an die Stub-Klasse zurückgegeben, wodurch dann auch der Client erfährt, ob die Bankleitzahl gültig ist.

Wie bereits erwähnt, muss sich der Clientprogrammierer, aber auch der Programmierer der Session Bean um diese Kommunikation nicht kümmern. Es sollte lediglich deutlich werden, dass der Client bei einer verteilten Anwendung niemals direkt mit einer Beanklasse arbeitet, sondern immer nur mit Stellvertreterklassen. Wenn der Stub seinen Aufruf an den Skeleton schickt, wird die Grenze zum Applikationsserver überschritten. Dieser ist an jeglicher Kommunikation beteiligt und sorgt dafür, dass die benötigte Beaninstanz auch zur Verfügung steht. Außerdem startet er eine Datenbanktransaktion, wenn es gewünscht ist, und vieles mehr.

Auch eine Session Bean kann als Client fungieren. Stellen wir uns eine komplexe Anwendung vor, an die unter anderem eine Bankleitzahl übergeben wird und die diese auf dem Server prüfen muss. Die komplexe Anwendung selbst ist auch eine Session Bean, die verschiedene Möglichkeiten hat, die Methode `blzGueutig()` aufzurufen. Die erste besteht darin, einfach eine Instanz von `BankBean` zu erzeugen und die Methode zu benutzen. In diesem Fall kann man aber nicht von einem Client sprechen. Die zweite Möglichkeit ist, sich an den eigenen Server zu wenden, um Zugriff auf den Bankleitzahldienst zu bekommen. Bezieht man sich dabei auf das Remote-Interface, kann es wieder zu einer verteilten Architektur kommen, weil dann die gerufene Bean in einem anderen Rechner laufen kann, der beispielsweise Teil eines gemeinsamen Clusters ist. In dieser Konstellation ist die Session Bean ein echter entfernter Client.

Die dritte Möglichkeit für die komplexe Anwendung besteht darin, sich an den eigenen Server zu wenden und sich dabei aber auf das Local-Interface zu beziehen.

### 1.2.3 Local-Interface

Eine Beanklasse kann wahlweise auch über ein Local-Interface verfügen. Damit bietet sie ihre Dienste anderen Session Beans an, die sich im selben Applikationsserver befinden. Dabei kann es sich um dieselben Methoden handeln, die schon im Remote-Interface stehen. Es ist auch eine Session Bean denkbar, die nur ein Local-Interface hat.

Ruft jemand eine Methode an einem Local-Interface auf, handelt es sich um einen Client, der sich innerhalb des Applikationsservers befindet. Dennoch ist der Applikationsserver an dem Aufruf beteiligt und bietet wieder seine Funktionen an, um beispielsweise eine Subtransaktion zu starten oder einfach nur dafür zu sorgen, dass die benötigte Beaninstanz vorhanden ist.

## 1.3 JNDI

Verschiedentlich wurde erwähnt, ein Client meldet sich beim Applikationsserver an und teilt diesem mit, dass er an dem Bankleitzahldienst interessiert ist. Damit dies funktioniert, benötigt er einen Namensdienst, der dem Client eine Referenz auf ein entferntes Objekt geben kann. Bei dieser Referenz handelt es sich um die Instanz einer Stub-Klasse, wie sie bereits beschrieben wurde. Ein Namensdienst ist nichts anderes als ein Programm, das auf einem Server läuft und in dem unterschiedlichste Java-Instanzen unter einem jeweiligen Namen hinterlegt sind. Sucht ein Client eine solche Instanz unter einem bestimmten Namen, dann sagt man, der Client macht einen *Lookup*. Jeder Applikationsserver bringt so einen Namensdienst mit, auch JBoss. Sobald man JBoss gestartet hat, steht auch der Namensdienst zur Verfügung.

Ein Namensdienst kann gleichzeitig ein Verzeichnisdienst sein. Damit ist er leistungsfähiger, weil er jetzt die Namen in einer hierarchischen Struktur verwaltet, ähnlich der Verzeichnisstruktur einer Festplatte. Verschiedene Java-Instanzen können so unter selbem Namen in verschiedenen Verzeichnissen hinterlegt werden.

Ein solcher Namens- und Verzeichnisdienst wird über JNDI (Java Naming and Directory Interface) angesprochen. Das ist durchaus vergleichbar mit einer Verbindung zu einer Datenbank über die Schnittstelle JDBC. Das Dienstprogramm implementiert also JNDI und hört auf einem bestimmten Server auf einen vorgegebenen Port. Alles, was ein Client wissen muss, ist die Adresse dieses Servers und die Nummer des Ports. Damit ist es durchaus möglich, dass auf einem Rechner mehrere Namensdienste laufen, die alle auf verschiedene Ports hören.

Um eine Verbindung zu einem Namensdienst aufnehmen zu können, benötigt der Client eine Instanz der Klasse `javax.naming.InitialContext`, die das Interface `javax.naming.Context` implementiert hat. An den Konstruktor von `InitialContext` müssen herstellerabhängige Parameter übergeben werden, aus denen hervorgeht, welcher Namensservice wie aufzurufen ist. Diese Parameter werden entweder in Form einer `Hashtable` an den Konstruktor oder in Form von Umgebungsparametern (`-D...`) übergeben. Als weitere Alternative steht die Verwendung einer Properties-Datei (`jndi.properties`) zur Verfügung.

Welche Parameter mit welchen Werten zu versorgen sind, geht aus der Herstellerdokumentation hervor. In Listing 1.3 sind alle Angaben für JBoss enthalten. In diesem Beispiel werden die Parameter als `Hashtable` übergeben. Die Klasse `Properties` ist von `Hashtable` abgeleitet.

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
      "org.jnp.interfaces.NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES,
      "org.jboss.naming:org.jnp.interfaces");
p.put(Context.PROVIDER_URL, "jnp://localhost:1099");
Context ctx = new InitialContext(p);
```

**Listing 1.3:** Verbindungsaufbau mit JBoss

Zunächst werden die Klassen festgelegt, die für den Verbindungsaufbau verantwortlich sind. Schließlich folgt noch die Adresse, auf die der Namensservice hört. In vorliegendem Beispiel ist das der eigene Rechner (`localhost`) und dort der Port `1099`. Der Wert ist standardmäßig von JBoss vorgegeben und kann bei der Installation geändert werden. Bei einem professionellen Einsatz würde man `localhost` natürlich durch den Namen oder die IP-Adresse des Applikationsservers ersetzen.

Wurde der `Context` hergestellt, kann der Namensdienst nach einer Ressource befragt werden. Für das vorliegende Beispiel heißt deren Name `BankBean/remote`, wobei der Zusatz `remote` besagt, dass man sich auf das Remote-Interface beziehen will. Alternativ könnte man nach `BankBean/local` fragen, wenn die `BankBean` ein Local-Interface hätte und man sich selbst auf dem Applikationsserver befinden würde. Um nach der gewünschten Ressource Ausschau zu halten, wird die Methode `lookup()` der Klasse `Context` benutzt. Sie liefert als Ergebnis ein `Object` zurück, was in unserem Fall die Stub-Klasse ist, die das Interface `BankRemote` implementiert hat. Aus diesem Grund wird die Instanz auch auf `BankRemote` gecastet.

```
Object ref = ctx.lookup("BankBean/remote");
BankRemote br = (BankRemote) PortableRemoteObject
    .narrow(ref, BankRemote.class);
```

**Listing 1.4:** Lookup ausführen

Die gelieferte Referenz kann aber leider nicht einfach so auf `BankRemote` umgestellt werden. Dazu ist vielmehr die Methode `narrow()` der Klasse `javax.rmi.PortableRemoteObject` nötig. Das liegt daran, dass die gelieferten Objekte IIOP- und damit Corba-fähig sind, dies wird aber an entsprechender Stelle in Kapitel 3 noch genauer erklärt.

Wenn es bei den verschiedenen Methodenaufrufen nicht zu irgendwelchen Fehlern gekommen ist, kann der Client jetzt die Methode `blzGueلتig()` benutzen. Ein beliebiger Fehler ist es, sich bei dem Namen der Ressource zu vertippen. Kann der Namensdienst damit nichts anfangen, löst er eine `javax.naming.NamingException` aus, die von `Exception` abgeleitet ist und daher von einem Client unbedingt abgefangen werden muss.

### 1.3.1 Portable JNDI-Namen

Seit der Version 3.1 des EJB-Standards wurden die JNDI-Namen in ihrem Aufbau standardisiert. Bei den im vorangegangenen Abschnitt gezeigten Namen `BankBean/remote` handelt es sich um die JBoss-Version dieser Namen, die immer noch unterstützt und in diesem Buch auch teilweise noch verwendet wird. Ein allgemeingültiger, portabler JNDI-Name ist aber nach folgendem Muster aufgebaut:

```
java:global[/<app-name>]/<module-name>/<bean-name>
```

Der in eckige Klammern eingeschlossene `app-name` ist optional und wird nur dann verwendet, wenn die programmierte EJB-Anwendung mit Hilfe einer EAR-Datei (Enterprise Archiv) installiert wird. Ein solches Archiv besteht aus einem oder mehreren Modulen, die alle zusammen die eigentliche Anwendung repräsentieren.

In vorliegendem Beispiel soll die Session Bean `BankBean` angesprochen werden. Geht man davon aus, dass sie sich in einem Modul mit dem Namen `ErstesBean.jar` befindet, dann lautet der korrekte JNDI-Name `java:global/ErstesBean/BankBean`. Ist das Modul `ErstesBean.jar` obendrein Bestandteil des Archivs `ErsteApp.ear`, dann erweitert sich der JNDI-Name auf `java:global/ErsteApp/ErstesBean/BankBean`.

Eine Bean kann mehrere Remote-Interfaces haben. Um dann genau zu definieren, welches davon man ansprechen möchte, hängt man den voll qualifizierten Namen des Interface nach einem Ausrufezeichen an den JNDI-Namen in der Form

```
!<fully-qualified-interface-name>
```

an.

Für die Beispielbean würde das den Namen

```
java:global/ErstesBean/BankBean!server.kap01.BankRemote
```

ergeben.

Neben `java:global` gibt es noch `java:app` und `java:module`. Über diese Form der Adressierung ist es einer Bean möglich, eine andere Bean derselben Applikation (EAR-Datei) beziehungsweise eine andere Bean innerhalb derselben JAR-Datei anzusprechen. Die Syntax von `java:app` lautet

```
java:app/<module-name>/<bean-name>[!<fully-qualified-interface-name>],
```

die von `java:module`

```
java:module/<bean-name>[!<fully-qualified-interface-name>].
```

Laut eigener Aussage unterstützt die aktuell vorliegende Version von JBoss die portablen JNDI-Namen noch nicht komplett. Aus diesem Grund wird in diesem Buch ab und zu die alte Schreibweise verwendet, die neue Version ist aber auch immer mit angegeben und lediglich auskommentiert.

## 1.4 Ein einfacher Client

Im vorhergehenden Abschnitt wurde erklärt, wie ein Client eine Verbindung mit einem Applikationsserver aufnimmt, um an die Methoden des Remote-Interface einer Session Bean zu kommen. Davon abgesehen handelt es sich um ein ganz normales Java-Programm, das in Listing 1.5 komplett abgedruckt ist.

```
package client.kap01;

import java.io.*;
import java.util.Properties;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import server.kap01.BankRemote;

public class BankClient {

    public static void main(String[] args) {
        try {
            Properties p = new Properties();
            p.put(Context.INITIAL_CONTEXT_FACTORY,
                "org.jnp.interfaces.NamingContextFactory");
            p.put(Context.URL_PKG_PREFIXES,
                "org.jboss.naming:org.jnp.interfaces");
            p.put(Context.PROVIDER_URL, "jnp://localhost:1099");
            Context ctx = new InitialContext(p);

            Object ref = ctx.lookup("BankBean/remote");
            BankRemote br = (BankRemote) PortableRemoteObject
                .narrow(ref, BankRemote.class);

            BufferedReader din = new BufferedReader(new InputStreamReader(
                System.in));

            String blz;
            do {
                System.out.print("Bitte BLZ: ");
                blz = din.readLine();
                if (blz.length() > 0) {
                    if (br.blzGueltig(new Integer(blz)))
                        System.out.println("Die BLZ ist gültig");
                    else
                        System.out.println("Unbekannte BLZ");
                }
            } while (blz.length() > 0);
        } catch (NamingException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

**Listing 1.5:** Kompletter Client

Nachdem der Client die Verbindung mit dem Server hergestellt hat, fragt er den Benutzer in einer Schleife nach der zu prüfenden Bankleitzahl. Gibt der Anwender eine leere Zeichenkette ein, indem er sofort  drückt, ist das Programm zu Ende. Ansonsten wird die Prüfmethode aufgerufen und das Ergebnis präsentiert. Dem Methodenaufruf sieht man nicht an, dass er unter Umständen auf einem ganz anderen Computer abläuft.

Damit sind alle Bestandteile erläutert, die in einer verteilten EJB-Anwendung für den entfernten Methodenaufruf notwendig sind: das Clientprogramm, die Serverkomponente in Form einer Session Bean und das Remote-Interface, das als Schnittstelle zwischen den beiden dient. Jeder Aufruf funktioniert nach diesem Muster. Dabei kann die gerufene Business-Methode, wie sonst auch, unterschiedliche fachliche wie auch technische Exceptions auslösen, die dann jeweils an den Client zur Bearbeitung übergeben werden. In einer verteilten Architektur übernehmen die verschiedenen Komponenten unterschiedliche Aufgaben. Die Anwendungslogik liegt typischerweise komplett in den Session Beans, die sich auch um alle Datenbankzugriffe kümmern. Ein Client hat hier normalerweise die Aufgabe, mit dem Benutzer in Interaktion zu treten, seine Eingaben entgegenzunehmen und sie dem Server zur Verfügung zu stellen. Danach präsentiert er das ermittelte Ergebnis. Es gibt nun eine nicht enden wollende Diskussion darüber, ob es nicht sinnvoll sein kann, dass auch der Client über Logik verfügt, und was man überhaupt darunter versteht. Businesslogik gehört bestimmt nicht in den Client. Dafür hat man seinen Server, denn nur indem man seine Funktionalitäten bündelt, macht man sie leichter wartbar. Was ist aber mit Eingabeprüfungen? Solange sich diese auf so allgemeine Dinge wie reine Formatprüfungen beschränken, sind sie in Ordnung. Wenn das Format aber von fachlichen Gegebenheiten abhängt, gehört die Prüfung auf den Server. Aber wie schon erwähnt, dazu gibt es unterschiedliche Meinungen. Aus meiner Erfahrung heraus schafft man sich die größtmögliche Flexibilität, wenn die Clients nichts anderes machen, als Daten anzuzeigen und eingeben zu lassen.

## 1.5 Eine Entity Bean für die Datenbank

In diesem Kapitel soll ein möglichst komplettes Bild gezeichnet werden, hierzu gehören auch die Zugriffe auf die Datenbank. Für diese ist eine eigene Beanart, die so genannten Entity Beans, zuständig. Seit EJB 3.0 handelt es sich dabei um normale Java-Klassen, die in ihren Attributen die Spalten der zugehörigen Datenbanktabelle repräsentieren. Um auf diese zugreifen zu können, bietet die Klasse passende Getter- und Setter-Methoden. In Listing 1.6 ist die Entity Bean für die Tabelle BANK abgedruckt, über die festgestellt werden soll, ob es die genannte Bankleitzahl überhaupt gibt. Wichtig ist hier zunächst die Annotation `@Entity`, die aus einer gewöhnlichen Java-Klasse eine Entity Bean macht.

```
package server.kap10;

import javax.persistence.*;

@Entity
@NamedQueries({
    @NamedQuery(name = "Bank.findByBlz",
        query = "SELECT b FROM Bank b"
            + " WHERE b.blz = :blz ORDER BY b.bank")
})
@Table(name = "BANK")
public class Bank implements java.io.Serializable {
    private static final long serialVersionUID = 1L;

    private int satznr;
    private int blz;
    private String bank;
    private String pruefziffer;

    @Id
    @Column(name = "DATENSATZNR")
    public int getSatznr() {
        return satznr;
    }

    public void setSatznr(int satznr) {
        this.satznr = satznr;
    }

    @Column(name = "BLZ")
    public int getBlz() {
        return blz;
    }

    public void setBlz(int blz) {
        this.blz = blz;
    }

    @Column(name = "BANK")
    public String getBank() {
        return bank;
    }
}
```

```
public void setBank(String bank) {
    this.bank = bank;
}

@Column(name = "PRUEFZIFFER")
public String getPruefziffer() {
    return pruefziffer;
}

public void setPruefziffer(String pruefziffer) {
    this.pruefziffer = pruefziffer;
}
}
```

**Listing 1.6:** Die Entity Bean Bank

Sehen Sie sich die Entity Bean und die Session Beans genauer an, wird Ihnen auffallen, dass es hier keine SQL-Anweisungen zum Einfügen, Ändern oder Löschen einer Zeile in der Tabelle BANK gibt. Sie finden auch keine Methoden dazu. Tatsächlich stehen diese Funktionen aber alle komplett zur Verfügung. Es ist Aufgabe des Applikationsservers, die notwendigen Anweisungen zu generieren, wenn der Anwendungsentwickler die Instanz der Entity Bean entsprechend manipuliert. Dafür steht dem Server ein EntityManager zur Verfügung, der all diese Aufgaben übernimmt.

Soll beispielsweise eine neue Zeile in die Tabelle BANK aufgenommen werden, erzeugt der Programmierer eine Instanz der Entity Bean Bank, setzt die Attribute auf die einzufügenden Werte und übergibt die Beaninstanz an die Methode `persist()` der EntityManager-Klasse. Um mehr braucht er sich nicht zu kümmern. Insbesondere wird er nicht mit der Programmierung der zugehörigen SQL-Anweisung behelligt. Diese erzeugt der EntityManager aus den Angaben der Entity Bean. Das hat den ganz großen Vorteil, dass der Entwickler hier weniger Fehler machen kann und man für den Fall, dass sich an der Datenbankstruktur etwas ändern sollte, dies nur an einer zentralen Stelle ändern muss, nämlich an der Entity Bean. Sobald die geänderte Bean deployt ist, erzeugt der EntityManager angepasste Anweisungen.

Darüber hinaus bietet der EntityManager noch weitere Funktionen, auf die in Kapitel 10 eingegangen wird. Zieht man das eben Erläuterte strikt durch, kann man zu einer kompletten Anwendung kommen, in der sich keine einzige SQL-Anweisung finden wird. An dieser Stelle sollte erwähnt werden, dass es für Entity Beans eine eigene Abfragesprache gibt, die EJB QL genannt wird und von SQL hergeleitet ist. Hauptaufgabe dieser Sprache ist es, Anweisungen zu formulieren, über die bestehende Daten gefunden werden können. Das ist die einzige Aufgabe, die der EntityManager nicht automatisch übernehmen kann, außer man sucht in

einer Tabelle nach dem Primärschlüssel. In Listing 1.6 findet sich die Annotation `@NamedQuery`, mit der eine solche Abfrage formuliert und mit einem Namen versehen wird. Gesucht werden alle Banken, beziehungsweise Bankfilialen, mit derselben Bankleitzahl. Tatsächlich ist die Bankleitzahl nicht der Primärschlüssel dieser Tabelle, sondern eine fortlaufende Nummer.

Alle Abfragen, die nicht ausschließlich über den Primärschlüssel gehen, müssen wie auch immer formuliert werden. Das schränkt die Aussage, man wird keine einzige SQL- oder EJB-QL-Anweisung finden, etwas ein, aber wenn man alle Abfragen über `@NamedQueries` innerhalb der Entity-Bean-Klassen positioniert, führt das zu einer sehr aufgeräumten Architektur, die ideal zu warten ist.

Herr der Entity Beans ist der schon erwähnte `EntityManager`. Eine Session Bean braucht Zugriff auf ihn, um mit der Datenbank arbeiten zu können. In Listing 1.7 ist die Session Bean aus dem vorliegenden Beispiel noch einmal verkürzt abgedruckt, wobei nur die Stellen angegeben sind, die für das Verständnis und die Zusammenarbeit mit dem `EntityManager` notwendig sind.

```
import javax.persistence.*;

@Stateless
public class BankBean implements BankRemote {
    @PersistenceContext(unitName = "JavaEE")
    private EntityManager manager;

    public boolean blzGueltig(Integer blz) {
        Query query = manager.createNamedQuery("Bank.findByBlz");
        query.setParameter("blz", blz);
        if(query.getResultList().size() == 0)
            ...
    }
}
```

**Listing 1.7:** Verkürzte Session Bean

Über die Annotation `@PersistenceContext` kann sich die Bean-Klasse eine Referenz auf den `EntityManager` injizieren lassen. Wichtig ist hier eigentlich nur, dass das Attribut `manager` vom Applikationsserver auf diesem Weg versorgt wird. In der Methode `blzGueltig()` wird zunächst eine Query erzeugt, indem die Methode `createNamedQuery()` des `EntityManagers` aufgerufen wird. Der Name der Abfrage ist `Bank.findByBlz`, so wie er in der Annotation `@NamedQuery` in der Entity Bean hinterlegt wurde. Als Nächstes wird der benötigte Parameter `blz` gesetzt und dann die eigentliche Abfrage über `getResultList()` gestartet. Diese Methode liefert eine Instanz von `List` zurück, in der so viele Instanzen von `Bank` enthalten sind, wie die Treffermenge ausmacht. Für das vorliegende Beispiel

mussten diese nicht einzeln abgefragt werden, da das einzig Wichtige die Größe der Treffermenge ist. Diese wird über `size()` abgefragt.

Hätte die Aufgabe darin bestanden, an allen Beans dieser Treffermenge eine Änderung durchzuführen, hätte man sich Instanz für Instanz in einer Schleife geben lassen und die entsprechende Setter-Methode für die Wertänderung aufgerufen. Alles andere macht dann wieder der EntityManager. Er registriert, an welcher Entity Bean Änderungen gemacht wurden, und veranlasst spätestens kurz vor Ende der Datenbanktransaktion, dass diese Änderungen in die Datenbank geschrieben werden. Der Anwendungsentwickler hantiert nur noch mit Java-Instanzen und nicht mehr direkt mit der Datenbank. Kritiker sehen hier übrigens den größten Nachteil einer EJB-Anwendung. Sie meinen, der Applikationsserver könne die SQL-Anweisungen nie so optimal erstellen wie ein Programmierer, was schlecht für die Performance ist. Tatsächlich lässt sich dies nicht ganz von der Hand weisen. Allerdings entlastet der EJB-Standard den Programmierer ganz erheblich und er wird eine ganze Menge Fehler weniger machen. Auch der Horror, dass wir unsere ganze Anwendung nach dem Wort BANK durchsuchen müssen, weil wir an der Tabelle etwas ändern wollen und wir einfach hoffen, auf diesem Weg alle SQL-Anweisungen zu finden, gehört der Vergangenheit an. Wie bei jeder technologischen Weiterentwicklung gilt auch hier, dass man die neu gewonnene Funktionalität mit einer etwas schlechteren Performance bezahlen muss. Ich kann mich noch sehr gut an die Diskussion darüber erinnern, was diese neumodischen und vor allem langsamen relationalen Datenbanksysteme sollen, ein direkter VSAM-Zugriff sei doch in jedem Fall schneller. Das ist zwar richtig, trotzdem arbeitet heute jeder mit DB2 und Oracle und wie sie sonst noch alle heißen.

Gerade das Persistence Management, also die Arbeit mit der Datenbank, spielt in der EJB-Welt eine sehr große Rolle, weshalb ihm gleich mehrere Kapitel in diesem Buch gewidmet sind.

## 1.6 Weitere Beanarten

### 1.6.1 Komplexe (Stateful) Session Beans

Neben den zustandslosen Beans wie dem Bankleitzahlendienst kennt ein Applikationsserver auch zustandsbehaftete. Der einzige Unterschied zu den zustandslosen Beans besteht darin, dass sie über die Annotation `@Stateful` verfügen und sich in ihren Attributen verschiedenste Informationen merken können, die sich alle auf einen bestimmten Benutzer beziehen. Während Stateless Session Beans allen gemeinsam zur Verfügung stehen, gehören Stateful Session Beans immer einem Anwender alleine. Sind also gleichzeitig einhundert Benutzer angemeldet und mit einer zustandsbehafteten Bean verbunden, verwaltet der Applikationsserver auch einhundert solche Instanzen. Wer jetzt aber Angst hat, dass dem Server irgendwann der Speicher ausgehen könnte, der kann beruhigt sein. Aktuell nicht

verwendete Beans werden auf die Platte ausgelagert und beim nächsten Request von dort wieder aktiviert. Eine hohe Benutzeranzahl bei gleichzeitig geringem Hauptspeicher macht sich maximal durch schlechte Performance bemerkbar und dem lässt sich sehr leicht entgegenwirken, indem der Server mit mehr Speicher ausgerüstet wird (was ist da heute schon ein Gigabyte) und wenn das immer noch nicht reicht, lässt sich ein zweiter Server danebenstellen und aus beiden ein Cluster bilden. Schon können sich die nächsten tausend Benutzer anmelden und intensiv arbeiten.

Welche Art von Bean man benutzt, ist abhängig von der Frage, was Sie programmieren wollen. Einfache Dienste, die aus den unmittelbar übergebenen Parametern ein Ergebnis zaubern, werden in Stateless Session Beans angelegt. Soll dagegen eine Kommunikation über mehrere Schritte mit dem Benutzer erfolgen, sind Stateful Session Beans die richtige Wahl. Ein gutes Beispiel, das in diesem Buch auch vorgestellt wird, ist das Führen eines Warenkorbs bei einem Online-shop. Der Kunde wählt einen Artikel nach dem anderen und merkt diese für seine Bestellung vor. Eventuell wird noch etwas an der Bestellmenge korrigiert oder einzelne Artikel werden wieder gelöscht, bevor er an die virtuelle Kasse geht und bezahlt. In einer zustandsbehafteten Bean kann dieser Warenkorb problemlos verwaltet werden. Es handelt sich um eine beliebige `Collection`, in der die bestellten Artikel mit Mengenangaben hinterlegt werden. Da jede Stateful Session Bean zu einem bestimmten Benutzer gehört, kommen sich die einzelnen Besteller gegenseitig nicht in die Quere.

Den Stateful Session Beans ist ein komplettes eigenes Kapitel gewidmet (Kapitel 5), in dem auf alle Belange dieser Beanart ausführlich eingegangen wird.

### 1.6.2 Singleton Session Beans

Bei einer Singleton Session Bean handelt es sich wieder um eine zustandslose Bean, von der es pro VM maximal eine Instanz geben wird. Das bedeutet, dass sich alle Aufrufer diese Instanz teilen müssen, was impliziert, dass die Methoden einer solchen Bean möglichst kurzläufig sein sollten. Die Besonderheiten dieser Beanart werden in Kapitel 6 vorgestellt.

### 1.6.3 Asynchrone Session-Bean-Methoden

Ruft ein Client eine Methode an einer Stateless, Stateful oder Singleton Session Bean auf, dann wird er typischerweise so lange in seiner weiteren Ausführung blockiert, bis die Methode serverseitig komplett abgearbeitet und das Ergebnis an den Client zurückgegeben wurde. Man spricht hier von synchronen Methodenaufrufen. Seit der Version 3.1 des EJB-Standards ist es jetzt auch möglich, asynchrone Session-Bean-Methoden zu definieren. Ruft ein Client eine solche Methode auf, bekommt er die Steuerung sofort wieder zurück. Ein weiterer Methodenaufruf an der Session Bean ist aber erst möglich, wenn der erste Aufruf serverseitig komplett abgearbeitet wurde.

Bei den verschiedenen Beanarten wird jeweils genauer erklärt, was es für sie bedeutet, wenn sie asynchrone Methoden anbieten. An dieser Stelle soll der Hinweis auf diese Art der Methoden genügen.

### 1.6.4 Asynchrone Message-Driven Beans

Neben den zustandslosen und zustandsbehafteten Beans gibt es noch eine weitere Beanart, die für die Abarbeitung von Anwendungslogik vorgesehen ist: die Message-Driven Beans. Sie unterscheiden sich durch einen ganz entscheidenden Punkt von den anderen beiden, denn sie sind niemals direkt mit einem Client verbunden. Bei einer Stateless Session Bean, einer Singleton Session Bean oder einer Stateful Session Bean macht der Client einen Lookup, bekommt eine Stub-Klasse geliefert, die das entsprechende Interface implementiert hat, und ruft dann die einzelnen Business-Methoden auf. Bei einer Message-Driven Bean ist das ganz anders.

Um diese Beanart benutzen zu können, muss unser Client in der Lage sein, mit einem Message Service arbeiten zu können. Es gibt eine ganze Reihe solcher Nachrichtensysteme wie beispielsweise das Programm MQSeries von der Firma IBM, HornetQ von JBoss oder SonicMQ von der Firma Sonic, um nur einige zu nennen. Allen gemeinsam ist, dass sie den Java-Standard JMS (Java Message Service) implementiert haben, der wieder vergleichbar ist mit JNDI für einen Namensservice oder JDBC für eine Datenbank. Solche Nachrichtensysteme unterstützen also bestimmte Java-Aufrufe in einer fest vorgegebenen Form, so dass die Implementierung jederzeit gewechselt werden kann, ohne dass an der Java-Anwendung etwas zu ändern wäre. So zumindest die Theorie, in der Praxis unterscheiden sich die verschiedenen Systeme hinsichtlich ihrer Leistungsfähigkeit. Es kann durchaus sein, dass das eine oder andere System keinen Transaktionsschutz bietet, obwohl dies eigentlich durch den Standard gefordert wird.

Eine Message-Driven Bean ist eine Java-Klasse, die an einem Nachrichtenkanal lauscht, ob Nachrichten für sie vorliegen. Wenn ja, wird die Nachricht verarbeitet und ein entsprechendes Ergebnis produziert. Es gibt zwei verschiedene Möglichkeiten, wie der Kanal, über den die Nachrichten versendet werden, gebaut ist. Man unterscheidet hierbei *Topics* und *Queues*. An einem solchen Nachrichtenkanal können gleichzeitig mehrere Sender und Empfänger angemeldet sein. Er sammelt alle gesendeten Nachrichten und versucht, sie zuzustellen. Handelt es sich um einen Topic-Kanal, wird die Nachricht an alle Empfänger weitergeleitet, bei einer Queue hingegen nur an den ersten, der nachfragt. Das ganze Zusammenspiel wird zwar in Kapitel 7 noch ausführlich erklärt, an dieser Stelle soll aber deutlich werden, warum eine Message-Driven Bean niemals direkt mit einem Client verbunden ist. Zwischen Client und Server gibt es ein zusätzliches Trägermedium, das die Nachricht zunächst entgegennimmt, womit für den Client die Arbeit getan ist und er sich neuen Aufgaben stellen kann. Das soll heißen, der Client wird nicht

auf eine Antwort warten, weshalb es sich beim Aufruf einer Message-Driven Bean immer um einen asynchronen Aufruf handelt. Die Nachricht wird dann zur gegebenen Zeit, die möglichst kurz sein sollte, an die angemeldeten Empfänger übergeben, die aufgrund dieser einen Nachricht in der Lage sein müssen, ihren Job zu tun.

Was so eine Message-Driven Bean machen soll, kann sehr unterschiedlich sein. Eventuell erzeugt sie ein Druckdokument aufgrund der Daten, die der Nachricht mitgegeben wurden. Wenn das die Aufgabe sein soll, muss man aber unbedingt beachten, ein transaktionssicheres Nachrichtensystem zu verwenden, weil es sonst leicht passieren kann, dass das eine oder andere Schriftstück verloren geht oder auch zu viel produziert wird. Transaktionssicherheit gibt es nicht nur beim Zustellen der Nachricht, sondern auch beim Versenden. Hat ein Sender eine Nachricht abgestellt und kommt es bei ihm dann doch noch zu einem Rollback, tut das Nachrichtensystem so, als hätte es niemals etwas gehört. Auch das unterstreicht noch einmal die Asynchronität. Erst wenn die Transaktion des Senders beendet ist, gilt die Nachricht als versendet und wird weitergeleitet.

Was ist aber, wenn der Sender unbedingt auf eine Antwort wartet? Dann ist es in jedem Fall einfacher, keine Message-Driven Bean, sondern beispielsweise eine Stateless Session Bean einzusetzen. Wenn das aber absolut nicht geht, kann man als Ergebnis der Arbeit auch von einer Message-Driven Bean aus eine Nachricht versenden, auf die der entsprechende Client hört. Jede Nachricht kann mit einem Message Selector versehen werden, der beispielsweise eine eindeutige Nummer beinhaltet und über den ein ganz bestimmter Client in der Lage ist, aus allen Nachrichten diejenigen herauszufiltern, die auch wirklich für ihn bestimmt sind. Die Programmierung des Clients ist etwas aufwendiger. Nachdem er seine Nachricht gesendet hat, fällt er in einen tiefen Schlaf, aus dem er erst wieder geweckt wird, wenn für ihn eine Antwort vorliegt oder wenn eine bestimmte Zeitspanne vergangen ist. Es macht keinen Sinn, dass ein Client endlos wartet, nur weil der Applikationsserver aktuell ein Problem hat. Was dies aber für die gesendete Nachricht bedeuten soll, muss individuell programmiert werden.

Der Einsatz einer Message-Driven Bean ist immer dann am sinnvollsten, wenn der Client keine Antwort benötigt. Auch andere Session Beans können Nachrichten versenden, um dadurch eine unabhängige Verarbeitung anzustoßen. Die Nachricht selbst kann dabei nicht nur aus einfachem Text bestehen, sondern es lassen sich ganze Java-Container übermitteln, in denen alle Informationen enthalten sind, die der Empfänger für seine Arbeit braucht.

Eine nachrichtengesteuerte Bean funktioniert ansonsten wie eine zustandslose Bean. Es ist unsinnig, sich irgendwelche Informationen in den Attributen der Java-Klasse für den nächsten Aufruf zu merken, da jede Instanz dieser Beanart allen Sendern zur Verfügung steht. Der Applikationsserver wird dafür sorgen, dass genügend Instanzen zur Verfügung stehen, um alle anstehenden Nachrich-

ten verarbeiten zu können. Ist eine Bean mit ihrer Arbeit fertig, wird sie sofort mit der nächsten Nachricht versorgt. Von welchem Client diese stammt, ist dabei unerheblich.

Eine Message-Driven Bean kennt weder ein Remote- noch ein Local-Interface, weil sie keine Business-Methoden anbietet. Vielmehr verfügt sie nur über eine einzige Methode, an die die nächste Nachricht übergeben wird. Diese Methode trägt den Namen `onMessage()` und liefert `void`, also nichts, zurück.

### 1.6.5 Zeitgesteuerte Timer Beans

Der Begriff Timer Bean ist eigentlich falsch, weil es keine eigene Timer-Beanart gibt. Was es gibt, ist ein Timer Service, den ein Applikationsserver unterstützen muss und der für Message-Driven Beans, Singleton Session Beans und Stateless Session Beans eingesetzt werden kann.

Instanzen dieser Beanarten können sich beim Timer Service anmelden und darum bitten, das nächste Mal an einem bestimmten Tag und zu einer bestimmten Uhrzeit oder einfach nach einer vorgegebenen Zeitspanne wieder aufgerufen zu werden. Des Weiteren ist es möglich, sich danach in einem fest vorgegebenen, immer gleichen Intervall erneut ansprechen zu lassen. Dazu muss die Bean-Klasse über eine Methode verfügen, die eine Instanz der Klasse `Timer` als Parameter erwartet und die mit der Annotation `@Timeout` versehen ist, damit der Timer Service herausfinden kann, um welche Methode es sich handelt.

Sobald die Zeit gekommen ist, ruft der Applikationsserver die gewünschte Methode der Bean auf und übergibt ihr die Instanz des zugehörigen Timers. Über diese Instanz lassen sich Informationen zwischen den Aufrufen weiterleiten, weil ihr jede beliebige serialisierbare Java-Klasse mitgegeben werden kann.

Timer sind persistente Objekte. Das bedeutet, dass sie vom Applikationsserver irgendwo gespeichert werden müssen, bis sie abgelaufen sind. Damit überleben sie einen Neustart des Servers. Hätte ein Timer ausgelöst werden müssen, während der Server nicht gelaufen ist, wird das sofort nachgeholt, wenn der Server wieder läuft. Es ist aber nicht genau spezifiziert, was passiert, wenn ein Intervall-timer mehrmals nicht zum Zuge kam. Nachdem der Applikationsserver hochgefahren wurde, wird ein solcher Intervalltimer aber mindestens einmal ausgelöst und daraufhin wieder in dem eingestellten Intervall. Es könnte aber auch sein, dass er gleich mehrfach hintereinander aktiviert wird.

Ein Timer Service will etwas in bestimmten Zeitabständen unabhängig von einer konkreten Clientaktion durchführen. Vielleicht gibt es in der Anwendung eine Datenbanktabelle, in die Arbeitsaufträge geschrieben werden, die irgendwann als abgearbeitet gekennzeichnet sind, und man will erreichen, dass diese automatisch eine Woche nach Bearbeitung gelöscht und eventuell archiviert werden. Für diese Aufgabe kann ein Timer eingerichtet werden, der sich nachts alle 24 Stunden die-

ser Tätigkeit annimmt. Er selektiert alle bearbeiteten Aufträge, deren Bearbeitung mehr als eine Woche her ist, und löscht sie.

## 1.7 Bestandteile von Java EE

Die Enterprise Edition verbindet verschiedene Technologien zu einem gemeinsamen Standard. Jeder Applikationsserverhersteller muss all diese unterstützen, wobei der Standard sogar die Versionsnummern der einzelnen Teile genau vorgibt. Um einen Eindruck von der Leistungsfähigkeit dieses Standards zu bekommen, sind alle geforderten Technologien nachfolgend aufgeführt.

- EJB 3.1
- Servlet 3.0
- JSP 2.2
- EL 2.2
- JMS 1.1
- JTA 1.1
- JavaMail 1.4
- Connector 1.6
- Web Services 1.3
- JAX-RPC 1.1
- JAX-WS 2.2
- JAX-RS 1.1
- JAXB 2.2
- JAXR 1.0
- Java EE Management 1.1
- Java EE Deployment 1.2
- JACC 1.4
- JASPIC 1.0
- JSP Debugging 1.0
- JSTL 1.2
- Web Services Metadata 2.1
- JSF 2.0

- Common Annotations 1.1
- Java Persistence 2.0
- Bean Validation 1.0
- Managed Beans 1.0
- Contexts and Dependency Injection for Java EE 1.0
- Dependency Injection for Java 1.0

Wie Sie obiger Liste entnehmen können, sind beispielsweise Webservices ebenso fester Bestandteil wie JavaServer Pages (JSPs) oder JavaServer Faces (JSFs), die beide für die Erstellung von Browseroberflächen eingesetzt werden. Ein anderer wichtiger Teil ist die J2EE Connector Architecture, die genau vorschreibt, wie man EJB-Anwendungen mit anderen Systemen wie CICS (Customer Information Control System, ein Transaktionsmonitor auf Großrechneranlagen) oder einfach in C programmierten Anwendungen verbindet, die ebenfalls Datenbankzugriffe vornehmen und die man alle unter einen gemeinsamen Transaktionsschutz bringen muss.

Will man sich mit dem kompletten Java-EE-6-Standard beschäftigen, müsste man sich mit all den genannten Technologien vertraut machen, was den Rahmen eines solchen Buches bei Weitem sprengen würde, sofern man die Themen nicht nur oberflächlich behandeln möchte. Einen wirklich sehr guten Überblick, allerdings in Englisch, erhalten Sie im Java EE 6 Tutorial im Internet auf der Seite <http://download.oracle.com/javaee/6/tutorial/doc/>.

## 1.8 Verbotene Beanoperationen

Es gibt verschiedene Dinge, die ein Anwendungsentwickler nicht machen darf, wenn er Enterprise JavaBeans programmiert. Das liegt einfach daran, dass er sich in einer Serverarchitektur befindet und nicht alleine die Kontrolle über alle Ressourcen hat. Der Applikationsserver ist Teil jedes Methodenaufrufs und gibt die Kontrolle kurzfristig an die Business-Methoden ab, die bedenken müssen, dass bei Methodenende wieder in den Applikationsserver zurückverzweigt wird. So ist zum Beispiel das Starten und Beenden einer Transaktion Aufgabe des Servers, außer Sie programmieren entsprechende Beans, die diese Aufgabe selbst übernehmen wollen. Der gesamte Zugriff auf die Datenbank, das Anmelden bei ihr, die Verwaltung der erzeugten Entity Beans und vieles mehr wird von der Serverumgebung erledigt, in die jede Bean eingebettet ist.

- Eine Bean darf keine statischen Attribute benutzen. Programmiert man sie dennoch, passiert zunächst einmal nichts Besonderes. Es gibt weder eine Fehlermeldung noch kommt es sofort zu irgendwelchen Problemen. Diese Einschränkung ist also nicht technisch, sondern eher fachlich bedingt. Der Inhalt

eines statischen Attributs gilt für alle Instanzen der Bean-Klasse und Sie müssen unbedingt daran denken, dass später gleichzeitig mehrere Benutzer mit dem System arbeiten sollen. Es macht also absolut keinen Sinn, irgendwelche benutzerabhängigen Informationen in ein solches Attribut zu speichern, weil dann immer nur eine Person mit dem System arbeiten kann. Will man aber irgendwelche Datentabellen einmalig aus der Datenbank lesen und im Hauptspeicher für alle lesend zur Verfügung stellen, dann lässt sich dies durchaus über statische Attribute machen, man muss aber bedenken, dass diese innerhalb eines Clusters auf eine Java VM begrenzt sind. Jede VM innerhalb des Clusters muss sich also um ihr eigenes Caching kümmern.

- Eine Bean darf nichts versuchen, um die Abarbeitung verschiedener Threads zu synchronisieren, außer es handelt sich um eine Singleton Session Bean mit Bean-Managed Concurrency.
- In einer Bean darf nichts aus dem AWT oder Swing Package benutzt werden. Es ist auch nicht sinnvoll, in einem Server eine grafische Oberfläche anzeigen zu wollen. Wer, außer dem Systemadministrator, sollte sie sehen? Aber selbst das ist verboten. Es darf nicht sein, dass der Applikationsserver solange warten muss, bis irgendwer einen OK-Button drückt.
- Alle Klassen aus `java.io` sind verboten. Damit ist gemeint, dass eine Bean niemals eine klassische Datei oder ein Verzeichnis lesen oder schreiben darf. Denken wir wieder an den Clusterbetrieb. Hier kann es passieren, dass der erste Methodenaufruf auf Rechner A und der zweite auf Rechner B ausgeführt wird, die technisch völlig voneinander getrennt sein können. Lediglich der Applikationsserver weiß von ihnen. Was will man mit einer Datei, die auf Rechner A liegt, wenn man sich selbst plötzlich auf Rechner B befindet? Noch gravierender könnte die Tatsache sein, dass sechzig Benutzer über ihre Bean-Klasse gleichzeitig versuchen, etwas Sinnvolles in dieselbe Datei zu schreiben. Dabei kann nichts Vernünftiges herauskommen. Thread-Synchronisation ist im Übrigen auch verboten.
- Der Aufbau und die Verwaltung eigener Socketverbindungen sind auch nicht erlaubt. Das Problem ist, dass man auch hier wieder in das Threadmanagement des Servers eingreifen muss und alle Ressourcen, auch Netzwerkverbindungen, dem Server gehören.
- Es ist einer Bean nicht gestattet, Methoden per Reflection-API aufzurufen oder Klassen hinsichtlich ihrer Member zu untersuchen.
- Es darf kein eigener Class-Loader benutzt werden, weil nur der Applikationsserver darüber wacht, welche Klassen er anzieht.
- Es dürfen keine eigenen Threads gestartet oder irgendwelche Threads gestoppt werden. Macht man es dennoch, ist nicht klar, was passiert.

- Auch das Laden nativer Bibliotheken ist nicht erlaubt. Verhindern kann und wird das der Applikationsserver wiederum nicht, allerdings tragen gerade solche Bibliotheken oft zur Instabilität des Servers bei. Sollte es in der Firma einen zentralen Rechenkern geben, der in C programmiert ist und unter Windows als DLL vorliegt, wird man als Entwickler nicht darum herumkommen, ihn zu benutzen. Die ganze Mathematik in Java noch einmal neu zu programmieren, kann nicht das Ziel sein. Allerdings muss der Rechenkern dann sicherstellen, dass er auf jeden Fall mehrbenutzerfähig ist. Java und insbesondere ein Applikationsserver stellt alle Dienste vielen Benutzern gleichzeitig zur Verfügung. Da kann es nicht sein, dass sich diese beim Rechenkern in einer Reihe anstellen müssen. Thread-Synchronisation ist wie erwähnt nicht erlaubt. Ganz kritisch wird es, wenn der Rechenkern auf eine Datenbank zugreifen will und im schlimmsten Fall sogar schreibend. Passiert das, muss er über die J2EE Connector Architecture als Ressource Adapter angebunden werden, weil dann ein gemeinsamer Transaktionsschutz hergestellt werden kann. Ich bekomme jedes Mal Panik, wenn der Kunde im laufenden Projekt meint, er habe da noch eine DLL ...
- Die Referenz `this` ist ganz kritisch. Jede Bean läuft unter der ausschließlichen Kontrolle des Applikationsservers und den darf man auf keinen Fall umgehen. Daher gilt, dass `this` weder an Methoden übergeben noch als Ergebnis einer eigenen Methode geliefert werden darf. Zu jeder Session Bean gibt es ein Session Handle, das sogar in einer Datenbanktabelle gespeichert werden kann. Dieses Handle ersetzt die `this`-Referenz.

Wie schon erwähnt, sind viele dieser Einschränkungen architekturbedingt. Es ist aber notwendig, sich damit auseinanderzusetzen und bei der täglichen Arbeit immer daran zu denken, dass es mehrere Benutzer geben wird, die gleichzeitig mit dem System arbeiten, und sich ein Applikationsserver über mehrere physische Rechner in einem Cluster erstrecken kann. Diese beiden Aspekte gilt es zu berücksichtigen, wenn man auf die Idee kommt, eine besondere Funktionalität programmieren zu wollen, die so nicht im Standard enthalten ist.

## 1.9 Annotations

### 1.9.1 Verwendung

Annotations sind Anmerkungen, die man zusätzlich zu einer Klasse machen kann, durch die diese näher beschrieben wird. Annotations beinhalten keine Logik, sondern markieren eine Klasse oder nur ein Attribut oder eine Methode daraus. Ein Beispiel für eine solche Anmerkung wurde in diesem Kapitel bereits vorgestellt. Die Annotation `@Stateless` markierte eine Java-Klasse als zustandslose Bean. Aufgrund dieser Anmerkung schreibt der Java Compiler eine entsprechende Information in die übersetzte Java-Klasse, die dann mit Hilfe der Java-

Reflection-API zur Laufzeit ausgelesen werden kann. Der Applikationsserver untersucht alle ihm bekannt gemachten Klassen und fragt jede, ob sie eine Annotation wie `@Stateless` trägt. Wenn ja, wird die Klasse besonders behandelt, in diesem Fall als zustandslose Bean.

In Listing 1.8 ist die Syntax der Annotation `@Stateless` dargestellt. Wie Sie sehen, handelt es sich dabei um eine Art Interface. Dass es um eine Annotation geht, erkennt man an dem Zeichen `@` vor `interface`.

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Stateless {
    String name() default "";
    String mappedName() default "";
    String description() default "";
}
```

**Listing 1.8:** Die Annotation `@Stateless`

Eine Anmerkung kann selbst über Anmerkungen verfügen. Diese werden Meta-Annotations genannt. In Listing 1.8 sind das `@Target` und `@Retention`. Über die Meta-Annotation `@Target` wird festgelegt, wo die Annotation eingesetzt werden darf. Mögliche Werte sind `TYPE`, also auf Klassenebene, `FIELD` für Attribute oder `METHOD` für Methoden. Es gibt noch weitere Möglichkeiten, die aber eher selten benutzt werden. Die vorgestellte Annotation `@Stateless` kann also nur auf Klassenebene benutzt werden; schreibt man sie vor ein Attribut oder eine Methode, wird das zu einem Übersetzungsfehler führen. Es kann durchaus sein, dass eine Annotation einmal über mehrere Typangaben verfügt. Sie kann dann sowohl auf der einen als auch auf der anderen Ebene benutzt werden. Trägt die Annotation `@Target` dagegen keinen Parameter, kann diese Annotation auch nie für sich in einer Java-Klasse eingesetzt, sondern nur als Meta-Annotation in anderen Annotations benutzt werden.

Die Meta-Annotation `@Retention` steht in dem Beispiel auf dem Wert `RUNTIME`. Damit kann die Anmerkung zur Laufzeit ausgewertet werden. Setzt man hier den Wert `CLASS`, was der Defaultwert ist, wird zwar ein entsprechender Eintrag in der übersetzten Java-Klasse gemacht, er kann aber nicht abgefragt werden. Eine Annotation mit `@Retention(SOURCE)` wird vom Compiler komplett ignoriert. Damit wird lediglich der Quellcode mit einer Anmerkung versehen.

Jede Annotation kann beliebig viele Parameter haben. Bei den Typen dieser Parameter ist man nicht beschränkt. Es können normale Java-Klassen oder auch Tabellen von diesen verwendet werden. Auch eine Annotation als Typ ist erlaubt. In dem Beispiel sind die Parameter die drei Zeichenketten `name`, `mappedName` und `description`. Alle drei sind laut Definition mit einem Defaultwert versehen. Setzt man bei der Verwendung der Annotation `@Stateless` keinen dieser Werte, erhalten sie alle leere Zeichenketten. Fehlt die `default`-Angabe bei einem solchen Parameter, muss er bei der Verwendung unbedingt gesetzt werden.

Zeichnet man eine Java-Klasse mit der Annotation `@Stateless` aus und macht keine weiteren Angaben, wird als Name für die Bean der Klassename der Java-Klasse herangezogen. Will man einen abweichenden Namen definieren, so schreibt man beispielsweise `@Stateless(name="EigenerName")`. Damit ist der Parameter versorgt.

```
@Target(METHOD) @Retention(RUNTIME)
public @interface Timeout { }
```

**Listing 1.9:** Die Annotation `@Timeout`

In Listing 1.9 ist ein weiteres Beispiel für eine Annotation abgedruckt. Diese kann nur auf Methodenebene eingesetzt werden und kennzeichnet in einer Bean diejenige Methode, die vom Timer Service des Applikationsservers aufgerufen werden soll, wenn die Zeit um ist. Auch hier sieht man noch einmal sehr schön, wie solche Anmerkungen benutzt werden. Alle notwendigen Informationen, die über die reine Anwendungslogik hinausgehen, können auf diese Weise den Java-Klassen mitgegeben werden.

Früher war es notwendig, dem Applikationsserver über eine zusätzliche XML-Datei mitzuteilen, bei welchen Klassen es sich um welche Beanarten handelt. Diese Deployment Descriptors mussten gepflegt werden, was man entweder von Hand oder mit Tools wie XDoclet gemacht hat. Sie wurden schnell groß und unübersichtlich. Der Einsatz von Annotations stellt hier eine erhebliche Erleichterung für den Entwickler dar. Heute kann man auf Deployment Descriptors fast völlig verzichten. Sie enthalten kaum mehr Informationen.

Annotations können auch geschachtelt vorkommen. Ein Beispiel dafür findet sich in Listing 1.10. Über die äußere Annotation `@NamedQueries` (man beachte die Mehrzahl) kann eine ganze Liste von `@NamedQuery`-Anmerkungen vergeben werden. Das Beispiel stammt aus der Entity Bean aus Listing 1.6 und dient der Definition einer Datenbanksuche. Deren Name soll `Bank.findByBlz` lauten, der Parameter `query` enthält einen gültigen EJB-QL-Ausdruck.

```
@NamedQueries({
    @NamedQuery(name = "Bank.findByBlz",
        query = "SELECT b FROM Bank b"
        + " WHERE b.blz = :blz ORDER BY b.bank")
})
```

**Listing 1.10:** Beispiel für verschachtelte Annotations

Betrachten Sie die Entity Bean in Listing 1.6 weiter, finden Sie noch eine ganze Reihe zusätzlicher Annotations, mit denen die Klasse beschrieben ist. Neben `@Entity`, die etwas über den Beantyp aussagt, finden Sie die Angabe `@Table(name="BANK")`,

was darauf hinweisen soll, dass die zugrunde liegende Datenbanktabelle den Namen BANK hat. Dann fallen noch `@Id` (für die Kennzeichnung des Primärschlüssels) und `@Column` auf, über die zusätzliche Informationen zu jeder Datenbankspalte hinterlegt werden können. Meist beschränken sich diese Angaben auf den Spaltennamen, der nicht immer dem des Attributs entsprechen muss.

In den folgenden Kapiteln werden alle diese Annotations ausführlich erklärt. Gerade für die Kennzeichnung der Entity Beans gibt es eine ganze Reihe solcher Anmerkungen.

## 1.9.2 Programmierung

Es lassen sich auch eigene Annotations programmieren, um seine Klassen damit auszuzeichnen. Der Applikationsserver wird sich für diese nicht interessieren, aber eventuell ein selbst geschriebener Client. Will man beispielsweise die vorgestellte Entity-Bean-Klasse auch für den Transport der Daten zum Client benutzen und diesem zusätzliche Informationen für die Datenanzeige mitgeben, kann es Sinn machen, zu diesem Zweck eigene Annotations zu definieren. Ein entsprechendes Beispiel findet sich in Listing 1.11.

```
package client.kap01;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Datenfeld {
    String bezeichnung() default "";
    Datentyp datentyp() default Datentyp.TEXT;
}
```

**Listing 1.11:** Eine eigene Annotation

Die Annotation `@Datenfeld` kann auf Attributebene benutzt werden und kennt zwei Parameter, die beide weggelassen werden können. Der Parameter `bezeichnung` soll dabei aussagen, unter welchem Namen das zugehörige Feld auf der Oberfläche des Clients präsentiert werden soll. Fehlt die Angabe, wird einfach der Name des Attributs selbst benutzt. Der Parameter `datentyp` will Einfluss auf eine mögliche Feldeingabe nehmen. Der Wert ist mit `TEXT` vorgelegt, kann aber bei der Verwendung der Annotation überschrieben werden. Zur Verdeutlichung ist die Aufzählung `Datentyp` in Listing 1.12 zu finden. Alle Parameter werden in Form von Methoden programmiert, der Standardwert wird über das Schlüsselwort `default` zugewiesen.

```
package client.kap01;

public enum Datentyp {
    TEXT,
    NUMMER,
    DATUM
}
```

**Listing 1.12:** Die Aufzählung Datentyp

Jetzt können die einzelnen Attribute der Entity Bean mit entsprechenden Anmerkungen versehen werden. Wie das aussehen kann, ist in Listing 1.13 verkürzt dargestellt. Nur die Attribute `satznr`, `blz` und `bank` sind mit der Annotation `@Datenfeld` versehen, was bedeuten soll, dass nur diese auf einer Oberfläche anzuzeigen sind. Bei `bank` wurde weder `bezeichnung` noch `datentyp` vergeben, wodurch die Standardwerte gesetzt sind.

```
@Entity
public class Bank implements java.io.Serializable {

    @Datenfeld(bezeichnung="Datensatznummer", datentyp=Datentyp.NUMMER)
    private int satznr;
    @Datenfeld(bezeichnung="Bankleitzahl", datentyp=Datentyp.NUMMER)
    private int blz;
    @Datenfeld
    private String bank;
    private String pruefziffer;
    ...
}
```

**Listing 1.13:** Entity Bean mit eigenen Annotations

Die beste Anmerkung hilft aber nichts, wenn es nicht auch jemanden gibt, der sie auswertet. Zu diesem Zweck wurden in Java 5 die Klassen `Class`, `Method`, `Constructor` und `Field` um die Methode `getAnnotation()` erweitert, die als Parameter die Klasse der gesuchten Annotation erwartet und diese als `Object` im Ergebnis liefert. Besitzt das abgefragte Objekt die Annotation nicht, wird `null` geliefert. Man muss also gezielt nach den Anmerkungen fragen. Alternativ kann man sich aber auch eine Tabelle aller vergebenen Annotations über den Aufruf `getAnnotations()` geben lassen.

In Listing 1.14 ist ein Client mit dem Namen `AnnotationClient` abgedruckt, der sich beim Applikationsserver anmeldet und von dort eine Instanz von `Bank` abholt. Danach untersucht er die Attribute dieser Klasse auf die Annotation `@Datenfeld` und gibt aus, was er findet.

```
package client.kap01;

import java.lang.reflect.Field;
import java.util.Properties;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import server.kap01.BankRemote;
import server.kap10.Bank;

public class AnnotationClient {

    public static void main(String[] args) throws Exception {
        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
            "org.jnp.interfaces.NamingContextFactory");
        p.put(Context.URL_PKG_PREFIXES,
            "org.jboss.naming:org.jnp.interfaces");
        p.put(Context.PROVIDER_URL, "jnp://localhost:1099");
        Context ctx = new InitialContext(p);

        Object ref = ctx.lookup("BankBean/remote");
        BankRemote br = (BankRemote) PortableRemoteObject.narrow(ref,
            BankRemote.class);

        Bank bank = br.getBank();
        Field[] fields = bank.getClass().getDeclaredFields();
        for (Field field : fields) {
            Datenfeld datenfeld = field.getAnnotation(Datenfeld.class);
            if (datenfeld != null) {
                System.out.println("Feld mit Annotation gefunden: "
                    + field.getName());
                if (datenfeld.bezeichnung().length() != 0) {
                    System.out.println("\tDatenfeld.bezeichnung = "
                        + datenfeld.bezeichnung());
                }
                System.out.println("\tDatenfeld.datentyp = "
                    + datenfeld.datentyp());
            }
        }
    }
}
```

**Listing 1.14:** Client für die Auswertung von Annotations

Wenn man den Client startet, liefert er das in Listing 1.15 abgedruckte Ergebnis.

```
Feld mit Annotation gefunden: satznr
  Datenfeld.bezeichnung = Datensatznummer
  Datenfeld.datentyp = NUMMER
Feld mit Annotation gefunden: blz
  Datenfeld.bezeichnung = Bankleitzahl
  Datenfeld.datentyp = NUMMER
Feld mit Annotation gefunden: bank
  Datenfeld.datentyp = TEXT
```

**Listing 1.15:** Clientausgabe

Wenn man sich näher mit dem Beispiel beschäftigt, fällt auf, dass man sich zwar eine Instanz der Klasse `Bank` vom Server besorgt, danach aber dennoch die Klasse untersucht. Das wird durch die Angabe

```
Field[] fields = bank.getClass().getDeclaredFields();
```

deutlich. Auf die Instanzvariable wird die Methode `getClass()` angewendet und für diese dann die Methode `getDeclaredFields()`. Die Methode `getFields()` würde nur die öffentlich zugänglichen Attribute liefern und alle Felder der Klasse `Bank` sind `private`, deshalb `getDeclaredFields()`. Die Instanz wird bei der Übertragung vom Server auf den Client serialisiert und auf dem Client wieder zu einer richtigen Klasse zusammengesetzt. Dazu muss der Client Zugriff auf die entsprechende Klasse haben. Man hätte sich also das ganze Anmelden am Server und Abholen einer lebenden Instanz sparen können. Damit soll gezeigt werden, dass die Annotations zusammen mit ihren Parametern bereits zur Compilezeit interpretiert, ausgewertet und in die übersetzte Klassendatei geschrieben werden. Zur Laufzeit ändern sie ihre Werte nicht mehr. Und hier liegen auch die Grenzen der Annotations. Will man sie oder auch nur ihre Parameter dynamisch setzen, hat man keine Chance. Aus diesem Grund sieht der EJB-Standard auch heute noch den Einsatz von Deployment Descriptors vor (externe XML-Dateien für die Beschreibung einzelner oder aller Beans). Wenn man sie verwendet, kann man mit ihnen alle Annotations, die sich fest in den Klassen befinden, überschreiben. Das ist aber keine Standardfunktionalität von Java, sondern explizit durch die Applikationsserver so programmiert. Findet man eine Definition in einem Deployment Descriptor, dann überschreibt oder ergänzt diese die Annotations.

## 1.10 Generics

Mit Java 5 wurden die so genannten Generics neu in diese Programmiersprache eingeführt. Anders als die Annotations werden die Generics im EJB-Standard nicht direkt benutzt. Sie tragen aber erheblich zur Verbesserung des geschriebe-

nen Quellcodes bei, weil sie für deutlich mehr Typsicherheit sorgen. Daher findet sich ihre Anwendung in dem einen oder anderen Beispiel in diesem Buch. Um diese zu verstehen, genügt es, die Schreibweise von generischen Klassen zu verstehen, selbst programmieren muss man sie nicht.

Dazu ein einfaches Beispiel. Die Klasse `java.util.Vector` gibt es bereits seit dem JDK 1.0, also von Anfang an. Sie implementiert eine dynamisch wachsende Tabelle aus beliebigen Objekten. Um ein neues Element aufnehmen zu können, sieht die Klasse die Methode `add()` vor, die eine beliebige Instanz von `Object` erwartet. Da wie bekannt alle Klassen von `Object` abgeleitet sind, können sich die unterschiedlichsten Typen in einem solchen `Vector` sammeln. Will man dann auf ein bestimmtes Element zugreifen, ruft man die Methode `get()` auf, die einen Index erwartet und als Ergebnis eine `Object`-Instanz zurückliefert, und zwar genau das Element, das an dem genannten Index gespeichert ist. Worum es sich dabei handelt, muss der Programmierer wissen. Castet er das Ergebnis auf den falschen Typ – er nimmt an, dort sei ein `String` gespeichert, bekommt aber eine `Integer` geliefert –, führt das zur Laufzeit zu einem sehr unangenehmen Fehler, der durch Generics vermeidbar ist.

Das Geheimnis liegt einfach darin, den `Vector` zusammen mit einem bestimmten Typ zu initialisieren. Wenn man schreibt,

```
Vector v = new Vector();
```

dann handelt es sich nicht um einen generischen Typ, sondern um einen `Vector`, wie wir ihn bereits kennen und er eben beschrieben wurde. Sollen dagegen nur Zeichenketten in dem `Vector` verwaltet werden, so gibt man das beim Aufruf des Konstruktors entsprechend an.

```
Vector<String> v = new Vector<String>();
```

Der gewünschte Typ wird in spitze Klammern unmittelbar hinter den Klassennamen geschrieben. In einem auf diese Art erzeugten `Vector` können nun nur Instanzen von `String` oder, wenn es das gäbe, von `String` abgeleitete Klassen gespeichert werden. Der Versuch, an die `add()`-Methode dieses Containers eine `Integer` zu übergeben, führt bereits zu einem Übersetzungsfehler. Die `get()`-Methode liefert für diesen `Vector` auch immer nur `String` zurück, weshalb eine Umwandlung des Ergebnistyps nicht notwendig ist. Die beiden Zeilen

```
v.add("Das ist auch ein String");  
String erg = v.get(0);
```

können problemlos auf den generischen `Vector` angewendet werden.

In Kapitel 11 über das Arbeiten mit Entity Beans wird eine Klasse `Datengruppe` vorgestellt, die für den Datenaustausch zwischen Server und Client verantwortlich ist und unter anderem einen `Vector` aus Instanzen der ebenfalls selbst entwickelten Klasse `DatenElement` beinhaltet. Das ist ein durchaus sinnvolles Beispiel für die Programmierung eigener generischer Klassen.

Weiter müssen Sie in dieses an sich sehr umfangreiche Thema hier nicht einsteigen, um die Beispiele zu verstehen.