8 JDBC

Databases are part of nearly every major software system. Since the quantities of data are nearly always too large to be permanently kept in the main memory of a computer, and because individual computers are switched off (or crash) from time to time, this data must be stored in a storage system. This may simply be a file system, but if this data is to be accessed more than once at various points in time, it is usually more efficient to store the data in databases. Also (and in particular) distributed applications are often connected with persistent data storage. Here, their function often exceeds that of simple storage of data: databases are also well suited for the connection of different applications and hardware systems. Since databases have a long history of use and many older systems manage their data in this way, they also represent a possibility of connection between new and old systems. Databases also offer the possibility of remote access and thus remote data communication. Remote database access is probably the most widely used mechanism for distributed communication. Therefore, the next chapters of this book are dedicated to the subject of persistence.

Databases in distributed systems

Among existing database technologies, relational databases are currently the most important ones, although recently the significance of object-oriented and object-relational databases keeps increasing. However, due to the mere fact that enormous amounts of data already exist in relational databases, they will not lose their overwhelming importance in the near future. Also, from their very structure, many kinds of data can more easily be captured in relational databases than in object-oriented ones. They allow efficient management and access to

Relational databases

tables of data which, in practice, frequently occur in the form of tables in the first place. Object-oriented databases will be discussed in more detail in the next chapter.

In this chapter, we will present the connection of Java to relational databases. For this purpose a standard exists which has become quite popular, the Java Database Connectivity

JDBC

(JDBC). With JDBC it is possible to access nearly every relational database; and when it is connected to the Internet, this can even be done from (practically) anywhere. It is assumed that the reader has a certain basic knowledge of the functioning of relational databases and the SQL query language.

Development of JDBC

JDBC has been developed out of a series of different standards for relational database access. The most important one is ODBC (Open Database Connectivity). An important goal in the development of JDBC was to facilitate the usability of such a database interface and make it extremely simple for standard cases of application, while allowing at the same time the use of the full bandwidth of possibilities provided by today's relational databases. Advanced mechanisms such as precompiled queries and working with meta-data will only be discussed marginally in this book. JDBC is an integrated part of Java and belongs to the core of the language. Like Java, it has had its development: in the JDK 1.1, JDBC is contained in its version 1.0, while the new version of Java (Java 2, also known as JDK 1.2) already includes JDBC version 2.0, which introduces extensions and improvements, but the core has remained unchanged.

First, the general structure of JDBC will be introduced and the most important interfaces will be presented in more detail. Subsequently, JDBC will be demonstrated with the aid of an example.

8.1 The structure of JDBC

A Java application can establish simultaneous connections to several databases. To make this possible, two layers have been incorporated into JDBC, which help with he management of these connections and make access as simple as possible. For each type of database, a database driver must exist which takes on the direct communication with the database. These drivers are managed by the driver manager, where the drivers are registered once and then handled automatically. The general structure of JDBC is shown in Figure 8.1.

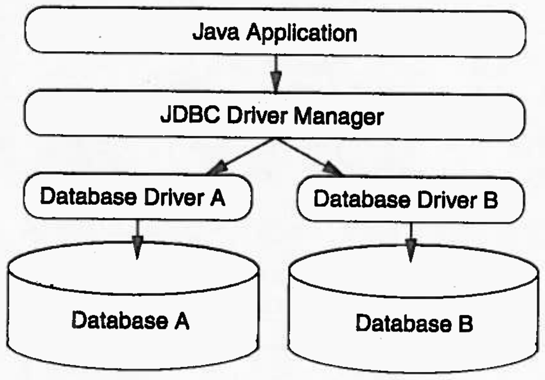


Figure 8.1 *The structure of JDBC.*

The driver manager

The driver manager is implemented via the DriverManager class and plays a central role inside the JDBC-API. This class is the interface to the different database drivers and establishes the connections with the databases, functioning as a link between the application and the databases. It is not instantiated, but used exclusively via class methods.

With DriverManager.registerDriver(), the drivers automatically register with this class, once they have been activated, and unregister with unregisterDriver(). Once a driver for one kind of database is known, this class does not need to be activated again, because the driver manager now recognizes itself which driver it must activate. Only when a driver is not yet known to the driver manager or has been unregistered, must it be registered again. DriverManager.getDrivers() can be used to find out which drivers are registered. The following paragraphs discuss the drivers in some more detail.

The database drivers

There are four variations of database drivers which chiefly differ by the kind of communication at protocol level. A driver can resort to the existing access protocols of ODBC, use proprietary database protocols, or run protocols standardized for JDBC. The drivers are therefore subdivided into the following four classes:

1. **JDBC/ODBC bridge**. This relatively slim conversion component allows ODBC drivers to be accessed by JDBC. This makes it possible to access every database that provides an ODBC interface even when it has no specific JDBC interface. Since JDBC has been developed on the basis of ODBC, only little frictional loss occurs, but it is slower than a direct link via one of the following alternatives. This driver can only be used by applications or by trusted applets, not by normal applets. It is provided free of charge by Sun and Intersolv.

2. **Platform-specific JDBC drivers**. These drivers convert JDBC calls directly into the proprietary calls of a database. As these are mostly not written in Java, but in C or C++, they cannot be loaded from the server into a browser and can therefore generally not be used for applets.

3. **Universal JDBC drivers**. In this solution, the driver is written in Java and communicates with a server component, also programmed in Java, via a database-independent protocol. The client part can automatically be loaded by an applet, so that nothing needs to be preinstalled on the client.

4. **Direct JDBC drivers**. These are drivers that work directly at the protocol level of the database. This solution is the fastest from the performance point of view, but has the disadvantage that the drivers are nearly exclusively supplied by the database manufacturers.

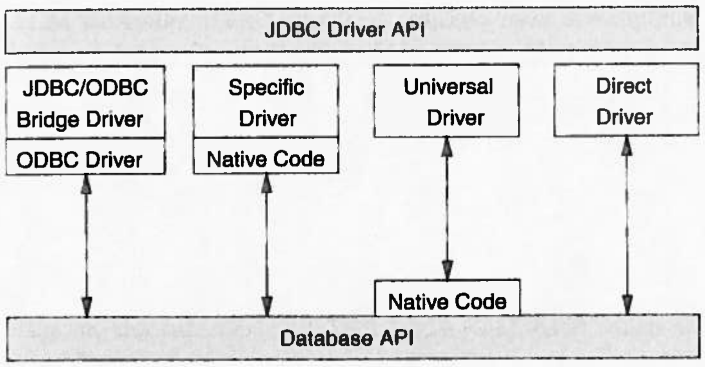


Figure 8.2
The different types of driver.

The two most frequently adopted solutions are the JDBC/ODBC bridge and the universal JDBC drivers. The bridge solution is the financially most economical one as this driver is available free of charge. However, it is also relatively slow and inflexible, while the universal drivers are highly flexible and, under performance aspects, comparable with the direct drivers.

Fortunately the decision as to which driver is going to be used is irrelevant for the programming of database applications. It has repercussions on costs and performance, but not on the interface to be used. However, as mentioned above, not all drivers are suitable for use in applets.

All drivers implement the java.sql.Driver interface, which a programmer has practically nothing to do with. A driver registers automatically with the DriverManager. It can then simply be called via a String that represents its class name or a URL. The JDBC-ODBC bridge driver used in the next example is included in the JDKs from version 1.1 onward.

```
// REGISTER DRIVER
try {
  Driver d = (Driver)Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
} catch (Exception e) {
   System.out.println(e)
}
```

Usually, it is even possible to do without a reference to this driver and simply use the abbreviated form:

Class.forName(sun.jdbc.odbc.JdbcOdbcDriver)

Everything else is controlled via the driver manager.

8.2 Establishing a connection with a database

For a normal sequence of queries, the course of a database application making use of JDBC looks as follows: the central class of JDBC is the DriverManager. This establishes one or more connections to one database each (the Connection). Via the Connection, individual queries (Statement) are created and passed to the database, which returns a set of result lines, the ResultSet. The result lines are read one by one by the application and then processed. Inside one Connection, several Statements can be executed in the context of one transaction, and be confirmed with a commit() or rejected with a rollback(). As the last step, the Connection is closed. The relation between these classes is shown in Figure 8.3.

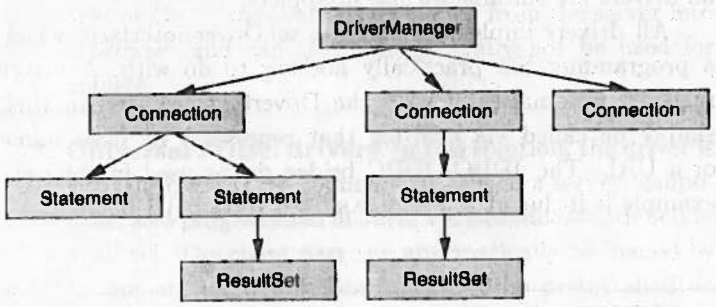


Figure 8.3 The driver manager handles several connections.

One of the most important functions of the driver manager is to establish and manage connections with one or more database(s). Each of these connections is handled by one object of the Connection class. The driver manager is passed a URL of a database, a user name, and a password, and will – if possible – return the required connection.

URLs for databases

As with other URLs, the URL for databases has the form: jdbc:<subprotocol>:<resourcename>, where the subprotocol specifies the driver to be used for the transmission. These subprotocol denominations are established by JavaSoft and can be reserved there by database or driver manufacturers. The JDBC/ODBC bridge, for example, has been assigned the subprotocol denomination odbc, so that an ODBC database can be accessed via jdbc:odbc://.... The resource name is the name of a database made available on a local computer or a computer on the Internet or Intranet. The database which will be used in the next examples can be accessed under the URL jdbc:odbc://sun.informatik.uni-hamburg.de/BulletinBoardDb or locally as jdbc:odbc://BulletinBoardDb.

```
// GET CONNECTION
Connection con;
try{
   con = DriverManager.getConnection(
    "jdbc:odbc://sun.informatik.uni-hamburg.de/BulletinBoardDb",userName,password);
}catch(Exception e){
   System.out.println(e);
}
```

8.3 Queries and answers

After a connection has been established, it can be used for putting queries to the database. In the development of JDBC, much care has been taken to make the solution of simple queries as easy as possible, while for more complex queries, users can put up with a more complex interface. Therefore, the queries are subdivided into three categories. For common SQL SELECT queries, the programmer can make use of a very easy-to-handle form, the Statement. For more complicated queries, the classes PreparedStatement and CallableStatement are available. Access to meta-data of a database, such as the structure of the tables, is made possible through a DatabaseMetaData class.

In order to put a normal SQL query, an object of the Statement class is created and passed to the database by means of the executeQuery()call which, as a parameter, contains an SQL query string. The database answers with an object of the ResultSet class, which contains a set of result lines which can be browsed through and read one by one.

```
Statement stmt = con.create();
String query="SELECT * FROM BlackBoard";
ResultSet results=stmt.executeQuery(query);
```

For database operations such as UPDATE, INSERT, and DELETE, Statement provides the executeUpdate() method, which returns a Boolean value.

```
String insert="INSERT INTO BlackBoard VALUES ('A new important message', 'Marko')";
Boolean bool=stmt.executeUpdate(insert);
```

The PreparedStatement class is an extension of Statement which is intended for queries that are often repeated in a similar form, with a view to optimization of performance. PreparedStatement makes it possible to pass a query containing placeholders to the database, which then precompiles the query. The placeholders are set with a set instruction, their actual values are passed at a later stage by means of an executeQuery() call.

```
PreparedStatement prepstmt = con.prepareStatement(
  SELECT * FROM BlackBoard WHERE user = ?);
prepstmt.setString(1,Marko);
ResultSet result1=prepstmt.executeQuery();
prepstmt.setString(1,Harald);
ResultSet result2=prepstmt.executeQuery();
```

CallableStatements are used in connection with *stored procedures* and provide an out parameter in addition to the in parameters.

Finally, the result which, after a query, is present as an object of the ResultSet class, must be processed. The next() method lets you browse through the lines of the result. The individual columns can be read by means of a get instruction which must know the type of the data field.

```
while (result1.next()) {
        String msg=result1.getString(Message);
        String user=result1.getString(User);
        System.out.println(Message from +user+ : +msg);
}
```

8.4 Example: a BulletinBoard

As an example we will now present an electronic bulletin board where messages are to be read from a database and displayed. For the sake of simplicity, a message (*Posting*) consists only of a headline and a text body and belongs to a group of subjects (*Category*). This simple structure can easily be extended to include an author or sender, the creation date and, where necessary, the date of expiry or something similar. For our purposes, however, a simple message will suffice.

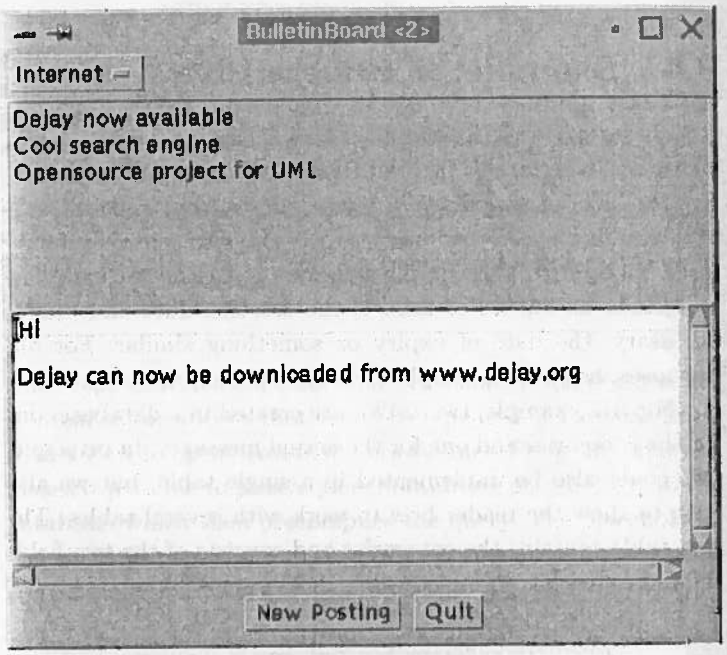
For this example, two tables are created in a database, one for the categories and one for the actual messages. In principle, this could also be implemented in a single table, but we also wish to show the reader how to work with several tables. The first table contains the categories and consists of the two fields ID and name.

ID	name
1	Giveaway
2	Search
3	Offer

The second table contains the messages and is subdivided into the fields ID, subject, and body; in an additional category field the category is indicated by its ID from the first table.

ID	subject	body	category
1	Vacuum cleaner	giveaway Hoover ...	1
2	Ford Thunderbird	used T-Bird, year ...	3
3	Cat escaped	striped cat, blue ...	2

The graphical interface of the application is split into two windows; the main window is generated by the BulletinBoardFrame class and displays the information from the database. An additional window, created by the NewPostingDialog class, is needed for the generation of new messages. Figures 8.4 and 8.5 show these two classes which are listed and explained in Appendix B.

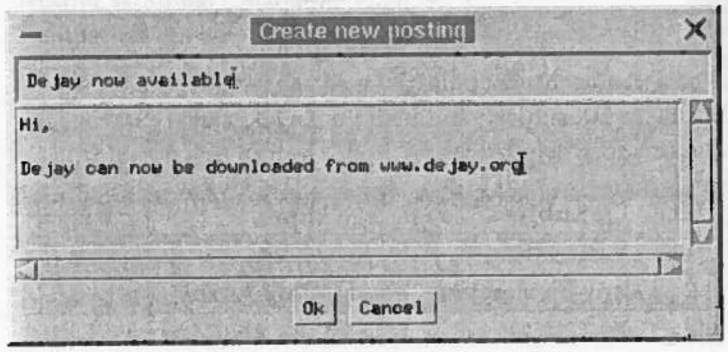


```
Dejay now available
Cool search engine
Opensource project for UML
```

```
Hi

Dejay can now be downloaded from www.dejay.org
```

Figure 8.4
The graphical interface of the BulletinBoard.



```
Dejay now available
```

```
Hi,

Dejay can now be downloaded from www.dejay.org
```

Figure 8.5
Input window for new messages.

```
import java

public class

  private B
  private C

  public B

    gui =

    // reg
    try {
      Cla
    } catc
      Syst
      Syst
    }

    // op
    try {
      con

    } catc
      Sys
    }

    try {
      Sta
       Re

      whi
        in
        S
        g
      }
    } catc
      Sys
    }
  }
```

```
import java.sql.*;

public class BulletinBoard {

  private BulletinBoardFrame gui = null;
  private Connection conn = null;

  public BulletinBoard() {

    gui = new BulletinBoardFrame("BulletinBoard", this);

    // register jdbc driver
    try {
      Class.forName("org.gjt.mm.mysql.Driver");
    } catch (ClassNotFoundException cnfe) {
      System.out.println("Can't load driver. Exiting.");
      System.exit(1);
    }

    // open connection to rdbms
    try {
      conn = DriverManager.getConnection(
          "jdbc:mysql://localhost:3306/jivs?user=jivs;password=jivs");
    } catch (Exception e) {
      System.out.println("Connection to RDBMS failed!");
    }

    try {
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery(
          "SELECT ID, name FROM categories ORDER BY name ASC");
      while (rs.next()) {
        int id = rs.getInt(1);
        String name = rs.getString(2);
        gui.addCategory(id, name);
      }
    } catch (SQLException e) {
      System.out.println("Reading categories failed: "+e);
    }
  }
```

```
public boolean handleQuit() {
   try {
      conn.close();
   } catch (SQLException e) {
      System.out.println("Closing connection failed");
   }
   return true;
}

public synchronized void insertNewPosting(String _subject, String _body,
      int _category) {
   try {
      Statement stmt = conn.createStatement();
      stmt.executeUpdate(
         "INSERT INTO postings VALUES
         (0,'"+_subject+"','"+_body+"',"+_category+")");
      reloadSubjects(_category);
   } catch (SQLException sqle) {
      System.out.println("Insert failed: "+sqle);
   }
}

public synchronized void reloadSubjects(int _category) {
   gui.clearSubjects();
   try {
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery(
         "SELECT ID, subject FROM postings WHERE
         category ="+_category );
      while (rs.next()) {
         int id = rs.getInt(1);
         String subject = rs.getString(2);
         gui.addSubject(id, subject);
      }
   } catch (SQLException sqle) {
      System.out.println("Select failed");
   }
}
```

```
pub
  tr



  }

  }
  re
}

pub
  B
}
}
```

8.5

As for
on JD
1997].
et al.
develo
Fo
omme

```
public String getBody(int _id) {
   try {
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery(
         "SELECT body FROM postings WHERE ID = "+_id);
      String body = null;
      if (rs.next()) {
         body = rs.getString(1);
      }
      if (body != null)
         return body;
      else
         return "";
   } catch (SQLException sqle) {
      System.out.println("Select failed");
   }
   return "";
}

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

8.5 References

As for Java in general, there exists a whole series of good books on JDBC. One of the most frequently quoted books is [Reese 1997]. A very good and sufficiently detailed book is [Hamilton *et al.* 1997], whose authors have been highly influential in the development of JDBC.

For a general introduction to relational databases, we recommend [Date 1995] or [Meier 1998].

9 Object-oriented databases

Disadvantages of relational databases

The relational databases discussed in the previous chapter are widely diffused in practical applications, their implementations are very reliable, highly optimized and perform very well. However, the fact that relational databases make use of data structures that do not quite fit into the object-oriented paradigm, creates a gap between programming language and database model which must be bridged by the programmer with relatively high effort. This fact is also known as *impedance mismatch*.

Relational databases have been developed after the theoretically founded relation model of Codd [Codd 1970], and the bridge towards programming languages has been made only at a later stage. The opposite way is followed by object-oriented databases: after the object-oriented paradigm had revealed itself as highly successful for programming, databases were developed that are compatible with the data structures employed, namely with objects. This database technology is much younger than the relational one, and scalability, stability, and performance have not yet reached a comparable level. However, the use of object databases makes programming substantially easier and permits a considerable increase of productivity. Therefore, object databases are currently given rather favorable odds.

Pros and cons of object-oriented databases

As opposed to relational databases, which possess a standardized query language, SQL, and standardized interfaces, such as ODBC and JDBC, the ways in which object databases are handled and accessed are still very dissimilar. Standardization attempts are being made, in particular by the ODMG (Object Database Management Group), a sub-organization of the

OMG, which is responsible for the standardization of object-oriented databases. However, these attempts have not yet been concluded and are only of limited significance. Therefore, one must today still look at individual products. The most important databases in this field come from companies such as Poet, Versant, Objectory, and ObjectDesign.

ObjectStore

In the following section, we will use the ObjectStore database by ObjectDesign. This has several reasons: on the one hand, ObjectDesign is the market leader in this field, so that the probability of finding this product in practice is extremely high. Also, ObjectStore's support for Java is very good and relatively easy to use. Third, ObjectStore exists in three different versions, the first of which, ObjectStore PSE, is available completely free of charge, while the second one is commercially available at moderate prices in the three-digit range. This version, ObjectStore PSE Pro, is compatible with the full product, so that programming remains the same. This version is designed for single-user systems, but otherwise provides the full range of features. The full product, ObjectStore, provides additional possibilities for multi-user operation and large amounts of data.

9.1 ObjectStore

When one thinks of databases, one usually imagines big server computers on which huge programs are running. That this must not necessarily be the case is proved by ObjectDesign with their ObjectStore Personal Storage Environment (PSE).

PSE

PSE is an object database with extremely low storage requirements, which is completely written in Java. It is well suited for storage and management of a larger number or persistent objects and, due to the fact that its memory requirements are so low, can even be used in applets. Nevertheless it provides very much of the functionality of fully-fledged databases, such as transactions and a query feature. For very large databases in which performance plays a major role, however, one should revert to the big brother.

PSE exists in two different variations. Initially, PSE was free of charge and was integrated, for example, into Netscape Navigator. Unfortunately, this version is no longer supported by ObjectDesign, but should still be available on the Web for quite a while. It has been designed for smaller amounts of data and single users, and does not allow parallel access to the database, but apart from these limitations, it offers nearly the complete functionality of an object-oriented database. Here the database is written to a file which is completely read into RAM when the database is opened. Therefore, PSE databases are subject to restrictions with regard to size and performance.

This version has been further developed by ObjectDesign to become ObjectStore PSE Pro which is, however, no longer free of charge. The cost, however, is more than reasonable. It can also be downloaded from ObjectDesign (currently in version 3) [ObjectDesign 1998] and be freely used for evaluation purposes for a period of 30 days. This version promises a good performance with data amounts of up to several hundred megabytes, and allows efficient indexing and garbage collection in the persistent storage. In this version, the entire database is not read, but only its currently relevant parts. This makes PSE Pro well suited even for large amounts of data. The following discussion is based on PSE Pro.

9.2 Accessing persistent objects

To access objects in a PSE Pro database, three preparatory steps are needed:

1. open a session,
2. open or create the database,
3. start the transaction.

Opening a session

All activities on a database are performed inside so-called *sessions* (Session). A session must first be created, then several threads of an application can participate in this session (join()). PSE supports exactly one session, while PSE Pro also supports several sessions opened in parallel.

```
Session session = Session.create(null,null);
session.join();
```

Opening a database

Before a database can be accessed, it must be opened. To do this, PSE/PSE Pro uses the Database.open() method. Databases have a name through which they are identified. Closed databases are kept on non-volatile storage media in the file system. In simple cases, the database file is located in the same directory as the Java application. Thus it can be simply identified via the file name. When the database is opened, you can specify whether it is to be accessed read-only or read and write. This is indicated in the second parameter via the READ_ONLY or UPDATE constants defined in the ObjectStore class.

```
Database db = Database.open("CounterDb.odb", ObjectStore.UPDATE);
```

Obviously, a database must exist in order to be accessed. If a database does not yet exist, it is created by means of the Database.create() method. Databases can be protected through the definition of access rights, in a similar way as, under Unix, files have read and write privileges for different user groups. For this purpose, constants such as ALL_READ, ALL_WRITE, and OWNER_WRITE are available. However, Java does not allow overriding of file system modes.

```
Database db = Database.create (
"CounterDb.odb", ObjectStore.ALL_READ|ObjectStore.ALL_WRITE);
```

Starting a transaction

The framework for all actions performed on the data of a database is the transaction. Inside a transaction, data can be accessed in reading or writing. When the transaction is terminated, all (modified) data items are written persistently in the database. But only then: prior to the end of a transaction, the changes are not visible to others. A transaction is started with begin() and terminated with commit(). Both methods have a parameter which controls the access to the data. During opening, an access mode must be specified. Transactions can be defined as read-only. Then, several such transaction can exist at the same time. For write access, however, only one single transaction may be open. This is indicated by means of the constants UPDATE and READ_ONLY.

```
Transaction trx = Transaction.begin(ObjectStore.UPDATE);
```

Root objects

Now, persistent objects can be accessed. Objects are persistent if and only if they are either a *root object* of the database or can be reached from a root object. Several root objects can be defined for one database. Usually, these are objects through which many other objects are referenced, such as a hash table or a vector. Unfortunately, in the JDK up to version 1.1, exactly these structures cannot be made persistent. For these cases, replacement structures exist which fulfill the same purpose and can be persistent. Here, ObjectStore provides special classes, such as OSHashtable and OSVector. However, with JDK 1.2 and the introduction of collections, this deficiency has been remedied.

A root object is created by means of createRoot(), which binds a name to an object. Root objects are the entry points in the persistent object graph and starting points of the navigation – the only way persistent objects can be reached.

```
OSHashtable counterHash = new OSHashtable();
db.createRoot("MyCounterHash", counterHash);
Counter accesses = new Counter();
accesses.set(42);
counterHash.put("AccessCounter", accesses);
trx.commit();
```

The name can later be used to access a root object, and subsequently all persistent objects, by applying the getRoot() method.

```
Transaction trx2 = Transaction.begin(ObjectStore.UPDATE);
counterHash = (OSHashtable) db.getRoot("CounterHash");
accesses = (Counter) counterHash.get("AccessCounter");
```

As already mentioned, transactions can be terminated by means of commit(), thus making the modifications persistent. All changes are stored simultaneously and undivided. All objects that can be reached from the root objects are made persistent. Thus the storage of persistent objects is carried out in a simple and transparent way for the programmer. It is also possible to reject the changes of a transaction and restore the state prior to the transaction. All changes to the data are either taken over completely into the database or not at all. To reject the changes, the abort() method is called.

```
accesses.increase();
if (accepted) {
  // the counter is incremented
  trx2.commit()
} else {
  // the counter is NOT incremented
  trx2.abort()
}
```

Access to persistent objects is (by default) only possible inside transactions. After a commit() has been executed, all persistent objects are made unreachable from the Java program. They are then called *stale*, and their values are deleted – obviously only in the program, and not in the persistent storage. When a stale object is accessed, the database system generates an exception. When a new access is to be made to one of these objects, a new transaction must be started and the required object must be reached from the root object. This is sensible, on the one hand, since in this way, changes made outside transactions cannot be written to the database, and the system cannot change into an inconsistent state. On the other hand, however, it is very useful to maintain certain states and be able to use them, for example, in new transactions. When the programmer so wishes, this can be specified through an additional parameter in the commit() method (as well as in the abort() method). This parameter too is an int for which again constants are available to express the required state. The constant RETAIN_STALE is equivalent to the default case without a parameter. With RETAIN_HOLLOW, references to persistent objects remain valid also outside the transaction, although any

Stale objects

access is still forbidden and generates an exception. In a new transaction, however, the objects can again be freely accessed. To be able to perform at least a reading access to such objects, RETAIN_READONLY is set. RETAIN_UPDATE even allows modifications, but as soon as the next transaction begins, these modifications are rejected.

One disadvantage of these procedures is that the corresponding objects cannot be collected by the garbage collector and thus encumber the main memory even when they are no longer needed. Therefore, one should from time to time terminate a transaction with RETAIN_STALE.

Terminating a session

Finally, the database must be closed and the session must be terminated. If this is not done, the database remains locked and cannot be opened again until the lock is deleted. In PSE, this lock is realized by creating and deleting an (otherwise empty) directory which bears the name of the database with the extension .odx.

```
db.close();
session.terminate();
```

Now that all of the individual steps have been explained, we will present a short but complete example. Here, objects of the already known Counter class are to be kept persistent.

```
import COM.odi.*;
import COM.odi.util.*;

public class CounterManager {

  public static void main(String[] args) {

    Session session = Session.create(null,null);
    session.join();
    Database db;
    OSHashtable counterHash = new OSHashtable();
    Counter accesses = new Counter();
    try {
      db = Database.open("CounterDb.odb", ObjectStore.UPDATE);
      Transaction trx = Transaction.begin(ObjectStore.UPDATE);
      db.createRoot("CounterHash",counterHash);
      accesses.set (42);
      System.out.println("Counter set to "+accesses.read());
      counterHash.put("AccessCounter",accesses);
      trx.commit();
    } catch (DatabaseNotFoundException e) {
      db = Database.create("CounterDb.odb",
          ObjectStore.ALL_READ|ObjectStore.ALL_WRITE);
    }

    Transaction trx2= Transaction.begin(ObjectStore.UPDATE);
    counterHash = (OSHashtable) db.getRoot("CounterHash");
    accesses  = (Counter) counterHash.get("AccessCounter");
    accesses.increase();
    System.out.println("Counter increased to "+accesses.read());
    trx2.commit(ObjectStore.RETAIN_READONLY);

    db.close();
    session.terminate();
  }
}
```

9.3 The postprocessor

With the mechanisms described above, ObjectStore offers a very easy-to-handle usage of persistence. To make this possible, however, classes must be postprocessed and prepared for persistence. This task is carried out by a postprocessor which reads the translated Java code and outputs it in modified form.

This postprocessor, named osjcfp (ObjectStore Java Class File Postprocessor), must be called after compilation with javac. It is controlled through a number of parameters, some of which are mandatory, while others are optional. In the easiest case, where the existing .class file is simply to be overwritten, the output must be redirected into the current directory (-dest .) and the postprocessor must be allowed to actually overwrite the existing file (-inplace). As the processor possesses two different modes, it also needs to be told whether the specified class

Mandatory parameters

is to be *persistence capable* (-pc) or only *persistence aware* (-pa). Here are the necessary steps, together with the output of the program:

```
sun> javac Counter.java CounterManager.java
sun> osjcfp -dest . -inplace -pc Counter.class
sun> java CounterManager
sun| Counter set to 42
sun| Counter increased to 43
```

To obtain information on an existing database, an additional tool is provided (together with several others): the osjshowdb command displays the class names and the number of objects in the database together with their size. For the database created in the above example, the following information is shown:

```
sun> osjshowdb CounterDb.odb
sun| Name: CounterDb.odb

sun| There is one root:
sun|  Name: CounterHash   Type: COM.odi.util.OSHashtable

sun| Segment: 0
sun| Size: 4111 (5 Kbytes)

sun|     Count  Tot Size  Type

sun|       2        64  COM.odi.util.OSHashtable
sun|       2        64  COM.odi.util.OSHashtableEntry
sun|       2        72  COM.odi.util.OSHashtableEntry[]
sun|       1        16  Counter
sun|       2        88  java.lang.Object[]
sun|       3       384  java.lang.String
sun|       1        16  java.lang.String[]
```

9.4 The BulletinBoard with ObjectStore

The following example shows how to make data persistent in an object-oriented database in a slightly more realistic application.

We will use the same example as with JDBC, the Bulletin-Board. Messages are sorted by categories and include a subject, the message body itself, plus a category number which stores the category to which the message belongs. For internal identification, the message is assigned an ID number. The complete object together with its access methods presents itself as follows.

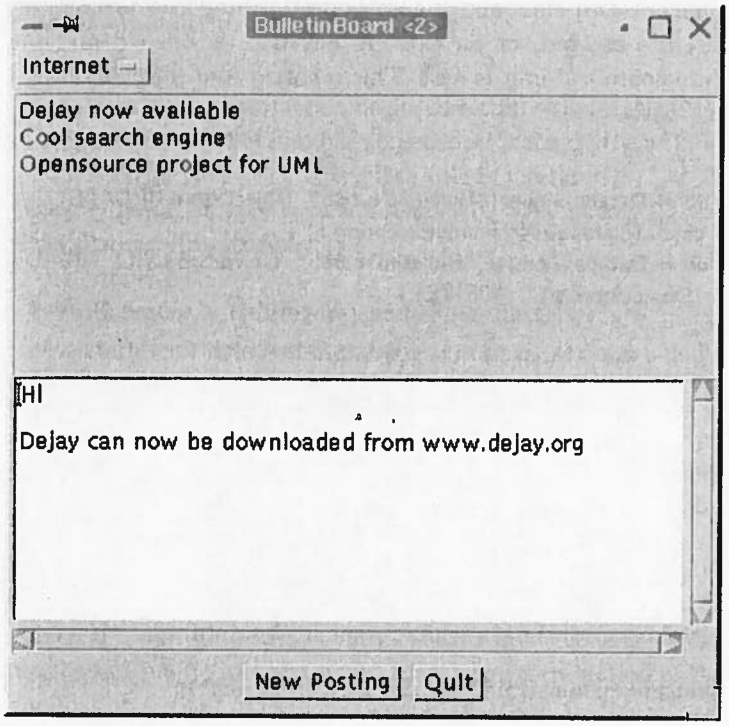
```
public class Posting {
  private int id;
  private String subject;
  private String body;
  private int category;
  public Posting() {
    id=0;
    subject = "";
    body = "";
    category = 0;
  }

  int getID() { return (id); }
  String getSubject() { return (subject); }
  String getBody() { return (body); }
  int getCategory() { return (category); }

  void setValues(int _id, String _subject, String _body, int _category) {
    id = _id;
    subject = _subject;
    body = _body;
    category = _category;
  }
}
```

Similarly to the JDBC example, there is also a simple class, Category, in which only the name of the category and the internal ID are defined together with the access methods.

Output and interaction with the user are performed via the same user interfaces defined in the BulletinBoardFrame class and listed in Appendix B. This generates the window shown in Figure 9.1.



Dejay now available
Cool search engine
Opensource project for UML

Hi

Dejay can now be downloaded from www.dejay.org

Figure 9.1 *The graphical interface of the BulletinBoard.*

When the user wishes to create a new message, pressing the "New Posting" button opens a new window in which the new message can then be entered.

The BulletinBoard class controls the storage of data and the communication with the BulletinBoardFrame, which in turn is responsible for the interaction with the user and the graphical presentation.

As root objects which house the postings and the categories, we will use two hash tables which have been adapted by ObjectDesign to suit the requirements of persistence (OSHashtable). Hash tables assign an entry a key (the hash code) through which it can again be reached.

```
// two root objects...
OSHashtable entryHash = new OSHashtable();
OSHashtable categoryHash = new OSHashtable();
```

Either, a database is generated in the constructor of the BulletinBoard class and the appropriate categories (search, offer) are created, or an existing database is opened and the data contained in it is read. This is carried out in the following try/catch clause:

```
try {
  db = Database.open("BulletinDb.odb", ObjectStore.UPDATE);
} catch (DatabaseNotFoundException e) {
  db = Database.create("BulletinDb.odb", ObjectStore.ALL_READ |
    ObjectStore.ALL_WRITE);}
```

Then a new transaction is generated in which the database can be accessed.

```
  Transaction trx = Transaction.begin(ObjectStore.UPDATE);
  db.createRoot("entryHash",entryHash);
  db.createRoot("categoryHash",categoryHash);
```

To set up the categories in the database, the put() method of the OSHashtable object is used. As soon as the transaction is terminated, this data is persistent in the database.

```
  categoryHash.put("1", new Category(1, "Search"));
  categoryHash.put("2", new Category(2, "Offer"));
  trx.commit(ObjectStore.RETAIN_HOLLOW);
}
```

The Objectstore.RETAIN_HOLLOW switch ensures that references to persistent objects in the database are maintained and can be used in subsequent transactions. The counterpart would be the default setting Objectstore.RETAIN_STALE which deletes the database cache, causing references to objects in the database to be lost.

The following listing shows a typical query to the database. All categories are read and passed to the BulletinBoardFrame for output. As the stored object is a hash table, querying the data is reduced to looking up this hash table. First, a new transaction is opened, then the getRoot() method is used to fetch a reference to the categoryHash object stored in the database. Via an iterator, all categories are read in a while loop and passed to the BulletinBoardFrame. After the current transaction has been terminated, the session can be exited to make it available to other threads.

```
Transaction trx = Transaction.begin(ObjectStore.READONLY);
categoryHash = (OSHashtable)db.getRoot("categoryHash");
Iterator categoryIter = categoryHash.values().iterator();
Category tempCategory = new Category(0, "no_matter");
while (categoryIter.hasNext()) {
  ecounter++;
  tempCategory = ((Category)categoryIter.next());
  int catID = tempCategory.getID();
  String name = tempCategory.getName();
  gui.addCategory(catID, name);
}
trx.commit(ObjectStore.RETAIN_HOLLOW);
session.leave();
```

The three methods (see source text) insertNewPosting(), reload-Subjects(), and getBody() control the further accesses to the database. Finally, it is important that, upon termination of the application, the functions close() of the database and terminate() of the session are called to terminate the transaction in an orderly way and to close the database.

```
// close database and terminate session
  public boolean handleQuit() {
    db.close();
    session.terminate();
    return true;
  }
```

The following listing shows the complete source code of the BulletinBoard class.

```
import COM.odi.*;
import COM.odi.util.*;

public class BulletinBoard {

  private BulletinBoardFrame gui = null;

  // define a session...
  Session session = Session.create(null,null);

  // ... and a database
  Database db;

  int ecounter = 0;

  // two root objects...
  OSHashtable entryHash = new OSHashtable();
  OSHashtable categoryHash = new OSHashtable();

  // constructor opens/creates the database and fills the GUI
  public BulletinBoard() {
    session.join();
    gui = new BulletinBoardFrame("BulletinBoard", this);
    Posting posting = new Posting();

    // try to open the database, otherwise create one and fill it with categories
    try {
      db = Database.open("BulletinDb.odb",ObjectStore.UPDATE);
    } catch (DatabaseNotFoundException e) {
      db = Database.create("BulletinDb.odb", ObjectStore.ALL_READ |
        ObjectStore.ALL_WRITE);
      Transaction trx = Transaction.begin(ObjectStore.UPDATE);
      db.createRoot("entryHash",entryHash);
      db.createRoot("categoryHash",categoryHash);
      // fill category
      categoryHash.put("1", new Category(1, "Search"));
      categoryHash.put("2", new Category(2, "Offer"));
      trx.commit(ObjectStore.RETAIN_HOLLOW);
    }
    // get all categories and pass them to the GUI
    Transaction trx = Transaction.begin(ObjectStore.READONLY);
    categoryHash = (OSHashtable)db.getRoot("categoryHash");
    Iterator categoryIter = categoryHash.values().iterator();
    Category tempCategory = new Category(0, "no_matter");

    // iterate over all categories...
    while (categoryIter.hasNext()) {
      tempCategory = ((Category)categoryIter.next());
      int catID = tempCategory.getID();
      String name = tempCategory.getName();
      gui.addCategory(catID, name);
    }
    trx.commit(ObjectStore.RETAIN_HOLLOW);
    session.leave();
    reloadSubjects(1);
  }
  // create new entry
  public synchronized void insertNewPosting(String _subject, String _body,
      int _category) {
    session.join();
    Transaction trx = Transaction.begin(ObjectStore.UPDATE);
    Posting tmpPosting = new Posting();
    ecounter++;
    tmpPosting.setValues(ecounter, _subject, _body, _category);
    Integer tmpecounter = new Integer(ecounter);
    entryHash.put(tmpecounter.toString(), tmpPosting);
    trx.commit(ObjectStore.RETAIN_HOLLOW);
    session.leave();
    reloadSubjects(_category);
  }
  // fetch the subjects from the database
  public synchronized void reloadSubjects(int _category) {
    session.join();
    gui.clearSubjects();
    Transaction trx = Transaction.begin(ObjectStore.READONLY);
    entryHash = (OSHashtable)db.getRoot("entryHash");
    Iterator postingIter = entryHash.values().iterator();
    Posting tempPosting = new Posting();
    ecounter=0;
    while (postingIter.hasNext()) {
      ecounter++;
      tempPosting = ((Posting)postingIter.next());
      int catID = tempPosting.getCategory();
      if (catID == _category) {
        String subject = tempPosting.getSubject();
        int id = tempPosting.getID();
        gui.addSubject(id, subject);
      }
    }
    trx.commit(ObjectStore.RETAIN_HOLLOW);
    session.leave();
  }

  // get the body for a posting
  public String getBody(int _id) {
    session.join();
    Transaction trx = Transaction.begin(ObjectStore.READONLY);
    Iterator postingIter = entryHash.values().iterator();
    Posting tempPosting = new Posting();
    String body = null;
    while (postingIter.hasNext()) {
      tempPosting = ((Posting)postingIter.next());
      int id = tempPosting.getID();
      if (id == _id) {
        body = tempPosting.getBody();
      }
    }
    trx.commit(ObjectStore.RETAIN_HOLLOW);
    session.leave();
    if (body != null) {
      return body;
    }
    else {
      return "";
    }
  }
```

```
  //
  pu

  }

  //
  pu

  }
}
```

9.5

In d
thus
sure
ever,
not b
cepts
as sc
cess
so-ca
whic
class
be n
ditio
sume
of pı
with
the a
also
eren
can
vate

```
  // close database  and terminate session
  public boolean handleQuit() {
    db.close();
    session.terminate();
    return true;
  }

  // main method
  public static void main (String[] args) {
    BulletinBoard bBoard = new BulletinBoard();
  }
}
```

9.5 Reactivation of objects

In distributed systems, individual components may fail and thus lose information about their state. Persistence can make sure that the data or the objects themselves are not lost; however, the consistency of remote references to these objects cannot be guaranteed. To solve this problem, further-reaching concepts are needed which can reactivate objects from a database as soon as they are accessed via remote references. This process is called activation. In Voyager, ObjectSpace provides a so-called *Activation Framework*, which does exactly this and which is particularly suited for use in distributed systems. The classes of the objects to be made persistent do not need to be modified or prepared in any way. Also, there are no additional precompilers or postcompilers. The mechanism presumes, however, the use of Voyager and, in particular, the use of proxies. This capability of Voyager can well be connected with an object-oriented database, as we will demonstrate with the aid of ObjectStore.

Activation Framework

When an object no longer exists in main memory, it can also no longer be referenced or called, not even via remote references. The Voyager Activation Framework ensures that this can nevertheless function. A persistent object can be reactivated, as soon or as long as a Voyager runtime environment is running under the appropriate port number on the original computer. Proxies pointing to such a persistent object can obviously still exist in other programs. When an object is called via such a proxy (which must contain some special activation information) and tries to contact the real object, it will notice that the object no longer exists. In this case, it sends the activation information to the Voyager runtime environment which then reactivates the object. Subsequently, the call can be executed as normal. For the developer, this mechanism, which is sketched in Figure 9.2, is transparent.

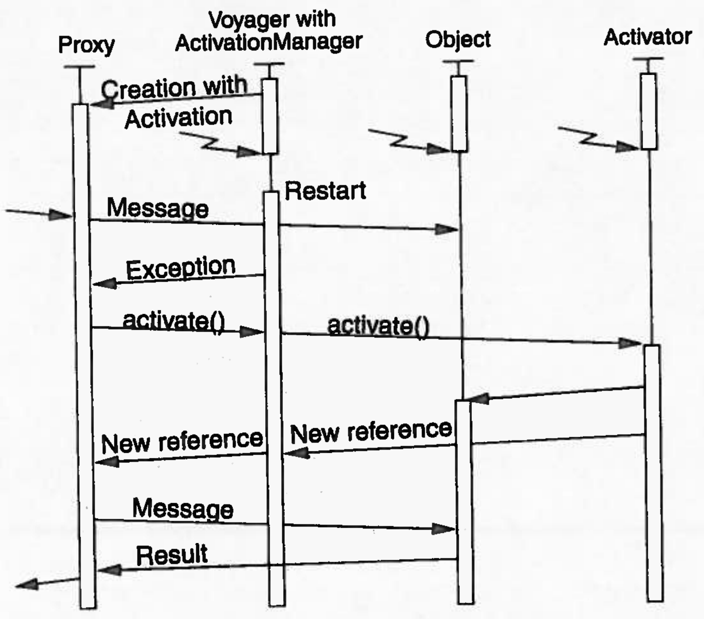


Figure 9.2
Activation of a persistent object.

Each Voyager runtime environment includes an Activation-Manager which manages a series of Activator objects. Each of these objects can in turn be connected with a database which either simply writes the serialized object into a file, or stores it in tables in a relational database, or makes it persistent in an object-oriented database. For each type of database, a class must exist which performs the mapping onto the database structure. This can either be written for arbitrary objects or developed for special applications. Thus, for example, an object can be mapped onto an existing table structure of a relational database. This mapping class must implement Voyager's IActivator interface and is then called Activator.

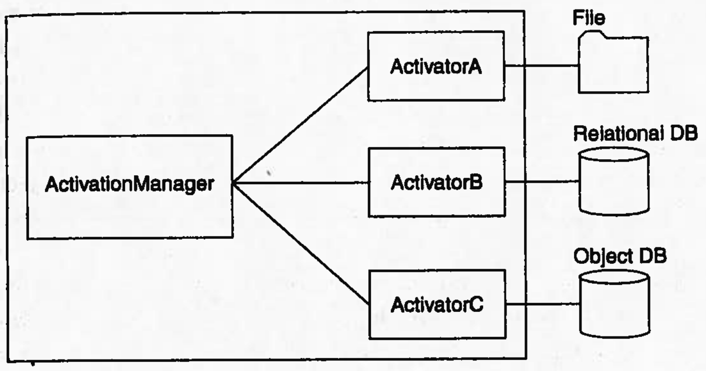


Figure 9.3
The Voyager ActivationManager.

The Memento

A new activator is registered with the manager via the call Activation.register(). This interface prescribes the methods getMemento() and activate(). The first method is passed a proxy for an object. It returns a string which is used to retrieve the object in the database. This string is called *Memento*, which means something like bookmark. The second method is passed this memento, with which the object is woken up from its persistence sleep and activated. Subsequently, it can again be accessed via the proxy.

These methods must, however, not be called by the developer, but by the ActivationManager. The developer must decide in the program, which objects are to be made persistent and which Activator should be used. Depending on the implementation, the Activator must be passed the name or the URL of the database and is then registered by calling Activation.register(). Subsequently, the Activator is told which objects are to be made persistent, by passing it a reference to an object or its proxy by means of the Activation.enable() method. Then all newly created proxies of this object receive the information required for the activation of the object. This consists of the URL of the program, the Memento of the object, and the class name of the Activator that has generated the Memento.

Upon the call of this method, the Activator writes the object into the database, before the object is destroyed by the garbage collector, or the program is exited, or the Voyager runtime environment is terminated by means of Voyager.shutdown(). With this, the object has become persistent and can be reactivated.

Let us demonstrate this mechanism with the aid of a simple example. We will again use the example of the bat and the ball, in which the Bat is supposed to remotely access an instance of the Ball type. The server on which the Ball object is located will, in the meantime, be shut down and restarted. The bat should access the ball once before and once after the server shutdown, reactivating the Ball at the second access. The following class, BallMachine, prepares the Ball for activation. As a database, ObjectStore PSE Pro is employed, for which a generic Activator class exists, the OSVoyagerActivator. Please note that at this point alternative Activators for different databases can be used as well; this merely requires a different instantiation of the Activator.

```
import com.objectspace.voyager.*;
import com.objectspace.voyager.activation.*;

public class BallMachine {

  public static void main(String[] args) {
    try {
      Voyager.startup("8000"); // start as server
      Ball ball = new Ball();
      Proxy.export(ball);
      Namespace.bind("ABall",ball);
      OSVoyagerActivator activator =
        new OSVoyagerActivator("testdb.odb");
      Activation.register(activator);
      Activation.enable(ball);
    } catch( Exception exception ) {
      System.err.println( exception );
    }
  }
}
```

The Ball class must only support serialization and is listed in the following code fragment. To simulate the termination of the server, Voyager is shut down by calling shutdown() at each call of the hit() method. Due to a flaw in the implementation of the OSVoyagerActivator, the ObjectStore session must also be terminated at this point with a call to terminate().

```
import com.objectspace.voyager.*;

public class Ball implements java.io.Serializable {

  public void hit() {
    System.out.println("Ball has been hit");
    COM.odi.Session.getGlobal().terminate();
    Voyager.shutdown();
  }
}
```

The Bat class receives a proxy for the Ball object through the name service where it has been registered by BallMachine and calls it twice. Between these two calls, the program stops and waits for user input, which gives enough time to restart the server which is, as we know, terminated with each call of hit().

```
import com.objectspace.voyager.*;

public class Bat {

  public void play(IBall ball) {
    System.out.println("Hitting the new Ball");
    ball.hit();
  }

  public static void main(String[] args) {
    try {
      Voyager.startup(); // start as client
      Bat bat = new Bat();
      IBall ball = (IBall) Namespace.lookup(
        "//vsyspc5.informatik.uni-hamburg.de:8000/ABall");
      bat.play(ball);
      System.out.println("Press key when Server restarts");
```

```
            System.in.read();
            bat.play(ball);
        } catch(Exception exception) {System.err.println(exception);}
        Voyager.shutdown();
    }
}
```

The server is restarted through the BallMachine2 class, which contains no information whatsoever about a Ball. Thus, when a new call of Ball occurs, this is reactivated exclusively through the Activation Framework and the activation information contained in the proxy.

```
import com.objectspace.voyager.*;
import com.objectspace.voyager.activation.*;

public class BallMachine2 {

    public static void main(String[] args) {
        try {
            Voyager.startup("8000"); // start as server
            OSVoyagerActivator activator =
                new OSVoyagerActivator("testdb.odb");
             Activation.register(activator);
        } catch( Exception exception ) { System.err.println(exception);}
    }
}
```

9.6 References

The field of object-oriented databases is still young and, as opposed to relational databases, the development of the market has not yet reached its peak. Therefore it is more difficult to find good, up-to-date literature. A book which lays a good foundation, explains the concepts, and discusses the differences against relational databases is [Heuer 1997]. Equally recommendable books are [Meier and Wüst 1997] and [Saake *et al.* 1997].

Information on ObjectStore can be found on the Web pages of the company [ObjectDesign 1998]. Also for other products, such as Poet, Versant, Objectory, and Gemstone, information can best be found on the Internet.

The description of Voyager and the Activation Framework is contained in the Voyager documentation [ObjectSpace 1998b]. Also JDK 1.2 by now includes an activation mechanism for RMI.