```
Random()
Random(long seed)
```

The first constructor creates a number generator that uses the current time as the starting value or seed value. The second form allows the user to specify a seed value manually. If the user initializes a Random object with a seed, then he/she can define the starting point for the random sequence. If the user uses the same seed to initialize another Random object, the user has to extract the same random sequence.

Task

create table ad(code number(5), name varchar(20), Join date(), salary number(10);

→ see the structure

desc (tablename)

→ insert data

insert into ad values(101, 'Don', '01-Jan-01', 2000);

For many task

insert into ad values(&code, '&name', '&Jdate', &salary);

For continuing with inserting

/ slash

Select * from ad; → To see the table

CHAPTER 20

Starting with JDBC

(2 copies)

About this Chapter: This chapter first gives us an introduction to JDBC. It teaches us how to connect to the database, create, alter and drop database objects and to perform manipulations with the database tables. It gives a brief description of the java.sql package, which is the most important package in JDBC. It concludes by giving an introduction to joins and transactions.

Objectives

- Introduction to JDBC
- Executing DDL and DML Commands
- Joins and Transactions

Introduction to JDBC

JDBC stands for **Java Database Connectivity**. It is a set of Java APIs used for executing SQL statements. This API consists of a set of classes and interfaces to enable programmers to write pure Java Database applications.

JDBC is a software layer that allows developers to write real client-server projects in Java. JDBC does not concern itself with specific DBMS functions. JDBC API (JDBC Application Programming Interface) defines how an application opens a connection, communicates with a database, executes SQL statements, and retrieves query results. Fig 20.1 illustrates the role of JDBC. JDBC is based on the X/OPEN Call Level Interface (CLI) for SQL.

Call Level Interface is a library of function calls that support SQL statements. CLI requires neither host variables nor other embedded SQL concepts that would make it less flexible from a programmer's perspective. It is still possible, however, to maintain and use specific functions of a database management system when accessing the database through a CLI.

for altering - Update Tablename Set Fieldname = ooo. where Customerid = 100;

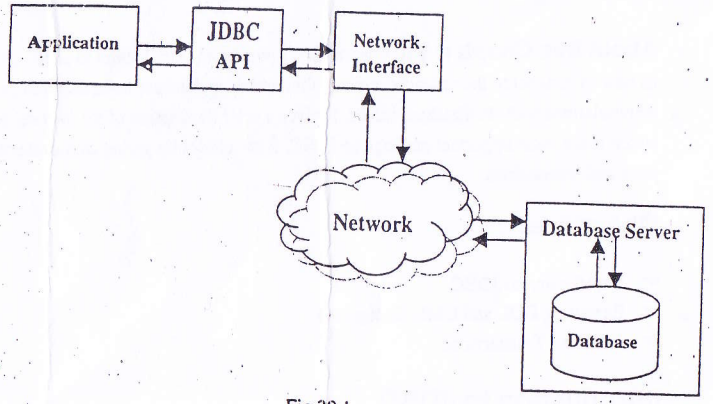

Fig 20.1

JDBC was designed to be a very compact, simple interface focusing on the execution of raw SQL statements and retrieving the results. The goal of creating JDBC is to create an interface that keeps simple tasks simple, while ensuring the more difficult and uncommon tasks are at least made possible. The following are the characteristics of JDBC:

- It is a call-level SQL interface for Java
- It does not restrict the type of queries passed to an underlying DBMS driver
- JDBC mechanisms are simple to understand and use
- It provides a Java interface that stays consistent with the rest of the Java system
- JDBC may be implemented on top of common SQL level APIs
- It uses strong, static typing whenever possible

Microsoft ODBC API offers connectivity to almost all databases on almost all platforms and is the most widely used programming interface for accessing relational databases. But ODBC cannot be directly used with Java programs due to various reasons enumerated in the JDBC Vs ODBC section. Hence the need for JDBC came into existence.

It is possible to access various relational databases like Sybase, Oracle, Informix, Ingres, using JDBC API. Using JDBC, we can write individual programs to connect to individual databases or one program that takes care of connecting to the respective databases.

Java and JDBC

The combination of Java with JDBC is very useful because it lets the programmer run his/her program on different platforms. Java programs are secure, robust, automatically downloaded from the network and Java is a good language to create database applications. JDBC API enables Java applications to interact with different types of databases. It is possible to publish vital information from a remote database on a web page using a Java

applet. With increasing inclination of programmers towards Java, knowledge about JDBC is essential.

Some of the advantages of using Java with JDBC are

- Easy and economical
- Continued usage of already installed databases
- Development time is short
- Installation and version control simplified

JDBC does the following three things

- Establish connection with a database
- Send SQL statements
- Process the results

There are two types of interfaces – low-level interface and high-level interface. While high level interfaces are user-friendly, low-level interfaces are not. JDBC is a low-level API interface, i.e., it is used to invoke or call SQL commands directly. The required SQL statements are passed as strings to Java methods.

Some of the current trends that are being developed to add more features to JDBC are embedded SQL for Java and direct mapping of relational databases to Java classes. Embedded SQL enables mixing of Java into a SQL statement. These statements are translated into JDBC calls using SQL Processor. In this type of direct mapping of relational database tables to Java classes, each row of the table becomes an instance of the class and each column value corresponds to an attribute of that instance. Mappings are being provided that makes rows of multiple tables to form a Java class.

JDBC Vs ODBC

The most widely used interface to access relational database today is Microsoft's ODBC API. ODBC performs similar tasks as that of JDBC and yet JDBC is preferred due to the following reasons:

- ODBC cannot be directly used with Java because it uses a C interface. Calls from Java to native C code have a number of drawbacks in the security, implementation, robustness and automatic portability of applications
- ODBC makes use of pointers which have been totally removed from Java
- ODBC mixes simple and advanced features together and has complex options for simple queries. But JDBC is designed to keep things simple while allowing advanced capabilities when required.

- ODBC requires manual installation of the ODBC driver manager and driver on all client machines. JDBC drivers are written in Java and JDBC code is automatically installable, secure and portable on all Java platforms from network computers to mainframes.
- JDBC API is a natural Java interface and is built on ODBC. JDBC retains some of the basic features of ODBC like X/Open SQL Call Level Interface.

Note A native method is a Java method, either an instance method or a class method, whose implementation is written, in another programming language like C.

JDBC Driver Models

JDBC supports two tier and three tier models.

Two tier model

In this type of model, Java applets /applications interact directly with the database. A JDBC driver is required to communicate with the particular database management system that is being accessed. SQL statements are sent to the database and the results are given to the user. This type of model is referred to as client / server configuration where user is the client and the machine that has the database is called as the server.

Three tier model

A middle tier is introduced in this model, which is used for various purposes. Some of the functions of the middle tier are:

- Collection of SQL statements from the client and handing it over to the database
- Receiving results from the database to the client
- Maintaining control over accessing and updation of the data

Middle tier until recently had been written in C, C++, which enable faster performance. With the introduction of optimizing compilers that translate Java bytecode into machine specific code, it is now possible to implement the middle tier in Java.

SQL Conformance

Structured Query Language (SQL) is the standard language used to access relational databases. Unfortunately, there are no standards set at present for it. For example, problems may arise due to the variations in different datatypes of different databases. JDBC defines a set of generic SQL type identifiers in the class Java.SQL.Types

Ways of dealing with SQL conformance

JDBC deals with SQL conformance by performing the following:

- JDBC API allows any query string to be passed through to an underlying DBMS driver. But there are possibilities of getting an error on some DBMS.
- Provision of ODBC style escape clauses.
- Provision of descriptive information about the DBMS using an interface, DatabaseMetaData.

The designation **JDBC Compliant** was created to set a standard level of JDBC functionality on which users can rely. Only the ANSI SQL 2 entry level supported drivers can make use of this designation. The conformance tests check for the existence of all classes and methods defined in the JDBC API and SQL entry level functionality.

Different Types of Driver Managers

JDBC contains three components: Application, Driver Manager and Driver. The user application invokes JDBC methods to send SQL statements to the database and retrieves results. JDBC driver manager is used to connect Java applications to the correct JDBC driver. JDBC driver test suite is used to ensure that the installed JDBC driver is JDBC Compliant. There are four different types of JDBC drivers. They are as follows:

- The JDBC-ODBC Bridge plus ODBC driver
- Native-API partly-Java driver
- JDBC-Net pure Java driver
- Native-protocol pure Java driver

The JDBC-ODBC Bridge plus ODBC driver

The JDBC-ODBC Bridge plus ODBC driver is a JavaSoft Bridge product that provides JDBC access via ODBC drivers. But as we have mentioned earlier, combining ODBC brings in a lot of drawbacks and limitations. Since the ODBC driver has to be installed on each client machine, it is not advisable to choose this type of driver for large networks.

Native-API partly-Java driver

Native-API partly-Java driver converts JDBC calls into calls on the client API for Oracle, Sybase, Informix or other DBMS. But some binary code has to be loaded on all clients like the bridge driver and hence is not suitable for large networks.

JDBC-Net pure Java driver

JDBC-Net pure Java driver translates JDBC calls into DBMS independent net protocol. A server again translates this protocol to a DBMS protocol. The net server middleware connects its pure Java clients to many different databases. The type of protocol in this middleware depends on the vendor.

Native-protocol pure Java drivers

These drivers convert JDBC calls to network protocols used by the DBMSs directly. Requests from client machines are made directly to the DBMS server.

Drivers 3 and 4 are the most preferred ways to access databases from JDBC drivers.

The java.sql package

We have seen that the JDBC API defines a set of interfaces and classes to be used for communicating with a database. These interfaces and classes are found in the java.sql package.

Interfaces in java.sql

Array

This interface was introduced in JDBC 2.0. It is used as the mapping in Java for the SQL type ARRAY. By default, an Array is a transaction duration reference to an SQL array. The methods in this interface are:

- getArray
- getBaseType
- getBaseTypeName
- getResultSet

Blob

This interface was introduced in JDBC 2.0. It is a mapping in Java of an SQL BLOB. An SQL BLOB is a built-in type that stores a Binary Large Object as a column value in a row of a database table. The methods of this interface are:

- getBinaryStream
- getBytes
- length
- position

CallableStatement

CallableStatements are used to call SQL stored procedures. A CallableStatement may return a ResultSet or multiple ResultSets. Escape syntax is used for procedures that return a parameter and those that do not return a parameter. This interface extends the PreparedStatement interface. The methods in this interface include the getXXX (where XXX stands for any datatype) methods and the following methods.

- registerOutParameter
- wasNull

Clob

The Clob interface was introduced in JDBC 2.0. It is a mapping in Java for the SQL CLOB type. An SQL CLOB is a built-in type that stores a Character Large Object as a column value in a row of a database table. The Clob interface provides methods for getting the length of an SQL CLOB value, for materializing a CLOB value on the client, and for searching for a substring or CLOB object within a CLOB value. The methods of the Clob interface are:

- getAsciiStream
- getCharacterStream
- getSubString
- length
- position

Connection

A Connection is a session in a specific database engine. Information such as database tables, stored procedures and other database objects may be obtained from a Connection with the getMetaData methods. Some of the important methods in this interface are:

- commit
- createStatement
- getAutoCommit
- isClosed
- isReadOnly
- prepareCall
- prepareStatement
- rollback
- setAutoCommit
- setReadOnly

DatabaseMetaData

The DatabaseMetaData interface provides information regarding the database itself, such as version information, table names, and supported functions. Many of the methods of this interface return lists of information in the form of ResultSet objects. Some of the important methods of this interface are:

- getCatalogs
- getColumns
- getConnection
- getDatabaseProductName
- getDatabaseProductVersion
- getDriverName
- getDriverVersion
- getMaxRowSize
- isReadOnly

Driver

Every driver must implement the Driver interface. The Driver interface is used to create Connection objects. When a Driver class is loaded, it should create an instance of itself and register it with the DriverManager. The following are the methods present in the Driver interface.

- acceptsURL
- connect
- getMajorVersion
- getMinorVersion
- getPropertyInfo
- jdbcCompliant

Ref

The Ref interface was introduced in JDBC 2.0. This interface is a reference to an SQL structured type value in the database. A Ref can be saved to persistent storage. A Ref is dereferenced by passing it as a parameter to an SQL statement and executing the statement. The only method in this interface is the getBaseTypeName method.

ResultSet

The ResultSet interface provides methods for the retrieval of data returned by a SQL statement execution. A ResultSet maintains a cursor pointing to its current row of data. The most often used methods, namely, getXXX and updateXXX methods are present in this interface. The other important methods present in this interface are:

- absolute
- afterLast
- beforeFirst
- cancelRowUpdates
- close
- deleteRow
- insertRow
- next
- previous
- wasNull.

ResultSetMetaData

The ResultSetMetaData interface is used for the collection of meta data information associated with the last ResultSet object. Some of the important methods of this interface are:

- getCatalogName
- getColumnName
- getColumnCount
- isNullable
- isReadOnly

SQLData

The interface SQLData was introduced in JDBC 2.0. It is used for the custom mapping of SQL user-defined types. The methods in this interface are called by the driver and are never called by a programmer directly. The methods of this interface are:

- getSQLTypeName
- readSQL
- writeSQL

SQLInput

It was introduced in JDBC 2.0. SQLInput is an input stream that contains a stream of values representing an instance of an SQL structured or distinct type. The programmer does not invoke this interface. Rather, the driver uses it. The readXXX methods (where XXX represents any data type) of this interface are used to read the attributes from the input stream. Another method in this interface is:

- wasNull

SQLOutput

The SQLOutput interface was introduced in JDBC 2.0 It is the output stream used for writing the attributes of a user-defined type back to the database. This interface is also used by the driver and the programmer does not invoke it directly. The writeXXX methods (where XXX represents any data type) of this interface are used to write data from the SQLData object to the output stream as the representation of an SQL user-defined type.

Statement

The methods of the Statement interface are used to execute SQL statements and retrieve data into the ResultSet. A Statement can open only one ResultSet at a time. Some of the important methods of this interface are:

- cancel
- close
- execute
- executeBatch
- executeUpdate
- getConnection
- getFetchSize
- getMaxRows

Struct

The Struct interface was introduced in JDBC 2.0. A Struct object contains a value for each attribute of the SQL structured type that it represents. By default, an instance of Struct is valid as long as the application has a reference to it. The two methods in this interface are:

- getAttributes
- getSQLTypeName

Classes in java.sql

Date

The Date class contains methods to perform conversion of SQL date formats and Java Date objects. It is a thin wrapper around a millisecond value that allows JDBC to identify this as an SQL Date. The Date class contains the following important methods:

- getHours
- getMinutes
- getSeconds
- setHours
- setMinutes
- setSeconds
- setTime
- toString
- valueOf

DriverManager

The DriverManager class is used to load and unload the drivers and establish the connection with the database. Since the DriverManager class attempts to load the driver classes referenced in the "jdbc.drivers" system property, the users to customize the JDBC Drivers used by their applications. The important methods of this class are:

- getConnection
- getDriver
- getLogStream
- println
- registerDriver

DriverPropertyInfo

The methods of the DriverPropertyInfo class are used for the insertion and retrieval of driver properties. It is useful for advanced programmers. This class inherits its methods from the java.lang.Object class.

Time

The Time class extends the Date class. It allows the JDBC to identify java.util.Date as a SQL Time value. The methods of this class are used to perform SQL time and Java Time object conversions. The methods available in this class are:

- getDate
- getDay
- getMonth
- getYear
- setDate
- setMonth
- setTime
- setYear

- toString
- valueOf

Timestamp

The Timestamp class also extends the Date class. It provides additional precision to the Java Date object by adding a nanosecond field. The methods of this class are:

- after
- before
- equals
- getNanos
- setNanos
- toString
- valueOf

Types

The Types class extends the java.lang.Object class. The Types class defines constants that are used to identify generic SQL types, called JDBC types. Its methods are inherited from the Object class.

Exceptions in java.sql

There are four exceptions in the java.sql class. Let us see each of them in detail.

BatchUpdateException

The BatchUpdateException extends the exception SQLException. The BatchUpdateException is thrown when an error occurs during a batch update operation. It provides the update counts for all commands that were executed successfully during the batch update.

DataTruncation

The DataTruncation extends the exception SQLWarning. The DataTruncation exception reports a DataTruncation warning or throws DataTruncation exception when JDBC unexpectedly truncates a data value.

SQLException

The SQLException extends the exception Exception. The SQLException provides information on a database access error. The information given in the exception includes a

string describing the error, a string describing the SQLState, the error code and a chain to the next exception.

SQLWarning

The SQLWarning extends the exception SQLException. The SQLWarning provides information on database access warnings and are chained to the object whose method caused it to be reported.

Steps for using JDBC

There are seven basic steps for using JDBC to access a database. They are:

- Import the java.sql package.
- Register the driver
- Connect to the database
- Create a statement
- Execute the statement
- Retrieve the results
- Close the statement and the connection

Before dealing about each of these steps in detail, let us consider the following case.

The Rhythm is a company selling audio cassettes. The company wants to maintain details of its customers and the transactions done with each customer. To automate these operations they contact **Pioneer Systems Ltd.** The team at Pioneer Systems Ltd. has decided to create the application in Java and use the Oracle database to store the information. JDBC is used to connect to Oracle. The description of the application will be given as we proceed with this chapter.

Import the java.sql package

The interfaces and classes of the JDBC API are present inside the package called as java.sql. Therefore, any application using the JDBC API must import the java.sql package in its code. Thus, the first line of the application should be

```
import java.sql.*;
```

Register the driver

To register the driver we make use of the method **registerDriver( )** whose syntax is as follows:

Syntax

```
DriverManager.registerDriver(Driver dr);
```

where `dr` is the new JDBC driver to be registered with the DriverManager.

Connect to the database

The next step is to connect to the database. The getConnection( ) method is used to establish the connection. The syntax of the getConnection method is given below:

Syntax

```
DriverManager.getConnection(String url, String user, String passwd);
```

where `url` is the database url of the form `jdbc:subprotodol:subname`, `user` is the database user and `passwd` is the password to be supplied to get connected to the database. The return value is a connection to the url.

Create a Statement

A statement can be created using three methods, namely, createStatement( ), prepareStatement( ) and prepareCall( ). The syntax of each of these is given below:

Syntax

```
cn.createStatement();
```

where `cn` is a connection object. This method creates and returns a Statement object for sending SQL statements to the database.

```
cn.createStatement(int rsType, int rsConcur);
```

where `cn` is a connection object, `rsType` and `rsConcur` denote the type and concurrency of the ResultSet, respectively. This method creates a Statement object that will generate ResultSet objects with the given type and concurrency.

```
cn.prepareStatement(String str);
```

where `cn` is a connection object and `str` is a SQL statement that may contain one or more IN parameter placeholders. This method creates and returns a PreparedStatement object for sending SQL statements with parameters to the database.

```
cn.prepareStatement(String str, int rsType, int rsConcur);
```

where `cn` is a connection object, `str` is a SQL statement, `rsType` is a result set type and `rsConcur` is a concurrency type. This method creates a preparedStatement object that will generate ResultSet objects with the given type and concurrency.


```
cn.prepareCall(String str);
```

where `cn` is a connection object and `str` is a SQL statement that may contain one or more IN parameter placeholders. This method creates and returns a CallableStatement object for calling database stored procedures.

```
cn.prepareCall(String str, int rsType, int rsConcur);
```

where `cn` is a connection object, `str` is a SQL statement, `rsType` is a result set type and `rsConcur` is a concurrency type. It creates a CallableStatement object that will generate ResultSet objects with the given type and concurrency.

SQL statements without parameters are normally executed using Statement objects. If the same SQL statement is executed many times, it is more efficient to use a PreparedStatement.

Execute the Statement

We have three methods to execute the statement. They are execute( ), executeQuery( ) and executeUpdate( ). Let us see the syntax each of these below:

Syntax

```
stmt.execute();
```

where `stmt` is a Statement object. This method returns a Boolean value and is used to execute any SQL statement.

```
stmt.execute(String str);
```

where `stmt` is a Statement object and `str` is an SQL statement. This method is used to execute an SQL statement that may return multiple results. The return value is a Boolean, which is true if the next result is a ResultSet and false if it is an update count, or there are no more results.

```
stmt.executeQuery();
```

where `stmt` is a PreparedStatement object. The method returns a result set generated by executing the query in stmt.

```
stmt.executeQuery(String str);
```

where `stmt` is an SQL Statement and `str` is the query to be executed. The method executes the SQL statement and returns a single ResultSet.

```
stmt.executeUpdate();
```

where stmt is a PreparedStatement object. This method executes the SQL statement in this PreparedStatement object. In addition, SQL statements that return nothing, such as SQL DDL statements, can be executed. The return value of this method is an int, which gives either the row count for INSERT, UPDATE or DELETE statements; or 0 for SQL statements that return nothing.

```
stmt.executeUpdate(String str);
```

where stmt is a Statement object and str is a SQL INSERT, UPDATE or DELETE statement or a SQL statement that returns nothing. This method executes the str and returns either the row count for INSERT, UPDATE or DELETE or 0 for SQL statements that return nothing.

Retrieve the Results

The results of the SQL statements (in particular, queries) are stored in a ResultSet object. To retrieve the data from the ResultSet we need to use the getXXX methods. These methods retrieve the data and convert it to a Java data type. There is a separate getXXX methods for each datatype. For example, getString( ) is used to retrieve a string value and getDate( ) is used to retrieve a date value. The getXXX methods take one argument, which is the index of the column in the ResultSet and retruen the value of the column. To move to the next row in the ResultSet we make use of the ResultSet.next( ) method.

Close the Statement and Connection

It is not absolutely necessary to close the connection. But, since open connections can cause problems, it is better to close the connections. The close( ) method is used to close the statement and the connection. The syntax is given below:

Syntax

```
stmt.close();
```

where stmt is the Statement object to be closed. This method releases stmt's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed. The return type is void.

```
cn.close();
```

where cn is the connection to be closed. This method releases cn's database and JDBC resources immediately instead of waiting for them to be automatically released. The return type is void.

To illustrate the above steps let us consider the case of The Rhythm company which was discussed earlier. Example 20.1 illustrates how to connect to the database. To confirm the connection, we display a message from the dual table in the database. (dual table is a pseudo table in the Oracle database).

Example 20.1

- Open a new file in the editor and type the following code:

Input/ Output

```
1.  import java.sql.*;
2.  class Customer {
3.  public static void main(String args[]) throws SQLException{
4.    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
5.    System.out.println("Connecting to the database...");
6.    try {
7.      Connection cn = DriverManager.getConnection
                ("jdbc:oracle:thin:@prakash:1521:orcl1","charu","charu");
8.      Statement st=cn.createStatement ();
9.      ResultSet rs= st.executeQuery("select 'Connected' from  dual");
10.     rs.next();
11.     String s=rs.getString(1);
12.     System.out.println(s);
13.     st.close();
14.     cn.close();
15.   } catch ( Exception ex) {
16.     System.out.println("The exception raised is : " + ex);
17.   }
18. }
19. }
```

- Save the program as **Customer.java**
- Compile the program using **javac Customer.java**
- Run the program using **java Customer**

The output is as follows:

```
Connecting to the database...
Connected
```

Caution The line numbers should not be entered in the code. They are given for explanation purposes.

Let us now explain Example20.1. Line 1 is the import statement that includes the java.sql package. Line 4 registers the OracleDriver after creating an instance of it. This is because the database to be used is Oracle 8. Line 6 connects to the database. The first argument of the getConnection method is the database url. In the url, **thin** is the JDBC driver, **prakash** is the database name, **1521** is the port on which the connection is to be established and **orcl1** is the system ID. Line 7 defines the statement object. Line 8 executes the statement using the executeQuery method.

It is to be noted that the contents of the ResultSet are retrieved using a pointer. This pointer is initially before the first row in the resultset. In order to move the pointer through the ResultSet, the next( ) method is used as in Line9. The getString( ) method in Line 10 retrieves the string data in the first column of the resultset. Since only one row is retrieved there are no more next( ) methods. The statement is closed in Line 12 and the connection is closed in Line 13.

A. Points to Ponder

State True or False

1. CallableStatements are used to call SQL stored procedures.
2. A Connection is a session in a specific database engine.

Fill in the blanks

1. ___ is a library of function calls that support SQL statements
2. Components of JDBC are _____, _____ and _____.
3. Every driver must implement the _____ interface

Executing DDL and DML Commands

Once the connection with the database is established, the user can start creating and working with the objects of the database. In this part we will be learning how to execute Data Definition Language (DDL) and Data Manipulation Language (DML) commands.

DDL Commands

The DDL Commands are create, alter and drop. Let us learn how to execute each of these. The create command is used to create database tables. Let us consider the case of "The Rhythm". The following are the tables needed by them:

Console . WriteLine
. show . cs .. . cs

Customer

CustId	Number(3)
CustName	varchar2(15)
Address	varchar2(30)

Product

Prodid	Number(3)
ProdName	varchar2(10)
Price	Number(5,2)
Stock_on_hand	Number(4)

Transaction

TranDt	Date
TranId	Number(3)
ProdId	Number(3)
CustId	Number(3)
Qty	Number(2)

The create statements for creating the above three tables are as follows:

```
Create table Customer (CustId number(3), CustName varchar2(15), Address
                                        varchar2(30));
Create table Product (ProdId number(3), ProdName varchar2(10), Price
                                        Number(5,2), Stock_on_hand Number(4));
Create table Transaction (CustId Number(3), ProdId Number(3), TranId
                                        Number(3), Qty Number(2), TranDt Date);
```

Example 20.2 illustrates the method of creating a table by giving the create command in the code itself.

Example 20.2

- Open a new file in the editor and type the following code:

Input/ Output

```
1.  import java.sql.*;
2.  class Customer {
3.    public static void main(String args[]) throws SQLException {
4.      DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```

```
5.    System.out.println("Connecting to the database...");
6.    Connection cn = DriverManager.getConnection
                ("jdbc:oracle:thin:@prakash:1521:orcl1","charu","charu");
7.    System.out.println("Connected to the database");
8.    Statement st = cn.createStatement ();
9.    try {
10.     st.executeUpdate("create table Customer(CustId number(3), CustName
                                varchar2(15), Address varchar2(30))");
11.     System.out.println("Table Customer Created");
12.   } catch (SQLException ex) {
13.     System.out.println ("The Exception raised is " + ex);
14.   }
15.   st.close();
16.   cn.close();
17. }
18. }
```

- Save the program as **Customer.java**
- Compile the program using **javac Customer.java**
- Run the program using **java Customer**

The following output is received on the screen.

```
Connecting to the database...
Connected to the database
Table Customer Created
```

In Line 10 of Example 20.2, the table is created using the executeUpdate() method. On successful completion, the "Table Customer Created" message is displayed. If an error occurs during the execution, then the catch clause is executed and the error is printed in Line 14.

In Example 20.3 the details of the connection are received from the user during execution and then the connection is established. It is used to create the Product and Transaction tables.

Example 20.3

- Open a new file in the editor and type the following code:

Input/ Output

```
1.  import java.sql.*;
2.  import java.math.*;
3.  import java.io.*;
4.  class UserConn {
5.  public static void main (String args []) throws SQLException,
                                                    IOException {
6.    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
7.    System.out.println ("Please enter the following information:");
8.    String user;
9.    String password;
10.   String database;
11.   user = readEntry ("User: ");
12.   int slash_index = user.indexOf ('/');
13.   if (slash_index != -1) {
14.     password = user.substring (slash_index + 1);
15.     user = user.substring (0, slash_index);
16.   }
17.   else
18.     password = readEntry ("Password: ");
19.  database = readEntry ("Database: ");
20.  System.out.flush ();
21.  try {
22.    Connection cn =DriverManager.getConnection ("jdbc:oracle:thin:@" +
                                        database,user, password);
23.    System.out.println ("Connected to the database");
24.    Statement st = cn.createStatement();
25.    st.executeUpdate("create table Product(Prodid number(3), ProdName
            varchar2(10), Price number(5,2), Stock_on_hand number(4));");
26.    System.out.println("Table Product Created");
27.    Statement stat = cn.createStatement();
28.    Stat.executeUpdate("create table Transaction(CustId Number(3), ProdId
            Number(3), TranId Number(3), Qty Number(2), TranDt Date);");
29.    System.out.println("Table Transaction Created");
30.    st.close();
31.    cn.close();
32.  } catch (Exception ex){
33.    System.out.println("The exception in creating the table is " + ex);
```