Listing 9-1 The Constructor

```
public conxtest()
{
   String url = "jdbc:odbc:customers";
   String userID = "jim";
   String password = "keogh";
   Statement DataRequest;
   ResultSet Results;

   try {
       Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

       Database = DriverManager.getConnection(url,userID,password);
   }

   catch (ClassNotFoundException error) {
       System.err.println("Unable to load the JDBC/ODBC bridge." + error);
       System.exit(1);
   }

   catch (SQLException error) {
       System.err.println("Cannot connect to the database." + error);
       System.exit(2);
   }

   try {
       String query = "Select * FROM customername";

       DataRequest = Database.createStatement();

       Results = DataRequest.executeQuery (query );

       Display (Results);

       DataRequest.close();
   }

   catch ( SQLException error ){
       System.err.println("SQL error." + error);
       System.exit(3);
   }

   setSize (500, 200);
   show();
}
```

Listing 9-2 The Display() Method

```
private void Display (ResultSet DisplayResults)throws SQLException
{
   Vector ColumnNames = new Vector();
   Vector rows = new Vector();
   boolean Records = DisplayResults.next();
   if (!Records )
   {
      JOptionPane.showMessageDialog( this, "End of data." );
      setTitle( "Process Completed");
      return;
   }
      setTitle ("Customer Names");

      try {
                ResultSetMetaData MetaData = DisplayResults.getMetaData();

                for ( int x = 1; x <= MetaData.getColumnCount(); ++x)
              ColumnNames.addElement (MetaData.getColumnName ( x ) );

          do {
             rows.addElement (DownRow( DisplayResults, MetaData ) );
                } while ( DisplayResults.next() );
```

```
            DataTable = new JTable ( rows, ColumnNames ) ;

            JScrollPane scroller = new JScrollPane ( DataTable ) ;
            getContentPane(). add ( scroller, BorderLayout.CENTER );

                  validate();
         }

         catch (SQLException error ) {
            System.err.println("Data display error." + error);
            System.exit(4);
         }
}
```

Listing 9-3 The DownRow() Method

```
private Vector DownRow ( ResultSet DisplayResults, ResultSetMetaData MetaData
           )
      throws SQLException
{
   Vector currentRow = new Vector();

   for ( int x = 1; x <= MetaData.getColumnCount(); ++x )
   {
      switch ( MetaData.getColumnType ( x ) ) {
         case Types.VARCHAR :
            currentRow.addElement( DisplayResults.getString ( x ) ) ;
            break;
      }
   }
   return currentRow;
}
```

Listing 9-4 The Disconnect() Method

```
public void Disconnect()
{
   try {
       Database.close();
   }
       catch (SQLException error) {
          System.err.println( "Cannot break connection." + error) ;
          System.exit(5);
   }
}
```

Listing 9-5 The Complete Listing

```
import java.sql.*;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import java.lang.*;

   public class conxtest extends JFrame
   {
      private JTable DataTable;
      private Connection Database;

      public conxtest()
      {
         String url = "jdbc:odbc:customers";
         String userID = "jim";
         String password = "keogh";
         Statement DataRequest;
         ResultSet Results;
```

```
      try {
                Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

                Database = DriverManager.getConnection(url,userID,password);
      }
      catch (ClassNotFoundException error) {
          System.err.println("Unable to load the JDBC/ODBC bridge." + error);
          System.exit(1);
      }
      catch (SQLException error) {
          System.err.println("Cannot connect to the database." + error);
                System.exit(2);
      }

      try {
          String query = "Select * FROM customername";
          DataRequest = Database.createStatement();
          Results = DataRequest.executeQuery (query );
          DisplayResults (Results );
          DataRequest.close();
      }
      catch ( SQLException error ){
          System.err.println("SQL error." + error);
          System.exit(3);
      }

      setSize (500, 200);
      show();
   }

private void DisplayResults (ResultSet DisplayResults)throws SQLException
{
   boolean Records = DisplayResults.next();

   if (!Records ) {
      JOptionPane.showMessageDialog( this, "End of data.");
   setTitle( "Process Completed");
   return;
   }
   setTitle ("Customer Names");
   Vector ColumnNames = new Vector();
   Vector rows = new Vector();

   try {
       ResultSetMetaData MetaData = DisplayResults.getMetaData();

       for ( int x = 1; x <= MetaData.getColumnCount(); ++x)
           ColumnNames.addElement (MetaData.getColumnName ( x ) );

       do {
          rows.addElement (DownRow( DisplayResults, MetaData ) );
       } while ( DisplayResults.next() );


       DataTable = new JTable ( rows, ColumnNames ) ;
       JScrollPane scroller = new JScrollPane ( DataTable ) ;
       getContentPane(). add ( scroller, BorderLayout.CENTER );
       validate();
    }
    catch (SQLException error ) {
       System.err.println("Data display error." + error);
       System.exit(4);
    }
}

private Vector DownRow ( ResultSet DisplayResults, ResultSetMetaData MetaData
          )
      throws SQLException
{
   Vector currentRow = new Vector();
   for ( int x = 1; x <= MetaData.getColumnCount(); ++x )
       switch ( MetaData.getColumnType ( x ) ) {
       case Types.VARCHAR :
          currentRow.addElement( DisplayResults.getString ( x ) ) ;
          break;
       }
   return currentRow;
```

```
   }

   public void Disconnect()
      {
      try {
          Database.close();
      }
      catch (SQLException error) {
          System.err.println( "Cannot break connection." + error) ;
          System.exit(5);
      }
   }

   public static void main ( String args [] )
   {
      final conxtest link = new conxtest();

      link.addWindowListener (
                new WindowAdapter() {
          public void windowClosing(WindowEvent WinEvent )
          {
          link.Disconnect();
          System.exit ( 0 ) ;
          }
      }

   );
   }
}
```

Listing 10-1 Creating a Table

```
import java.sql.*;

   public class createtb
   {
      private Connection Database;
      public createtb()
      {
         String url = "jdbc:odbc:customers";
         String userID = "jim";
         String password = "keogh";

         try {
                   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
                   Database = DriverManager.getConnection(url,userID,password);
         }
         catch (ClassNotFoundException error) {
             System.err.println("Unable to load the JDBC/ODBC bridge." + error);
             System.exit(1);
         }
         catch (SQLException error) {
             System.err.println("Cannot connect to the database. "+ error);
             System.exit(2);
         }

         Statement DataRequest;

         try {
                   String query = "Create Table address ( CustomerNumber
             CHAR(30), CustomerStreet CHAR(30), CustomerCity CHAR(30),
             CustomerZip CHAR(30))";
                   DataRequest = Database.createStatement();
                   DataRequest.executeQuery (query);
             DataRequest.close();
                   Database.close();
         }
         catch ( SQLException error ){
             System.err.println("SQL error." + error);
                   System.exit(3);
         }
      }

   public static void main ( String args [] )
   {
```

```
      final createtb link = new createtb();
      System.exit ( 0 ) ;
   }
}
```

Listing 10-2 Writing Clearer Code

```
try {
    String query = "Create Table address ( CustomerNumber CHAR(30)," +
                   "CustomerStreet CHAR(30)," +
                   "CustomerCity CHAR(30)," +
                   "CustomerZip CHAR(30))";
    DataRequest = Database.createStatement();
    DataRequest.executeQuery (query );
    DataRequest.close();
    Database.close();
}
```

Listing 10-3 Using NOT NULL

```
try {
          String query = "Create Table customers ( " +
                   "FirstName CHAR(30) NOT NULL," +
                   "LastName CHAR(30) NOT NULL)";
          DataRequest = Database.createStatement();
          DataRequest.executeQuery (query );
          DataRequest.close();
          Database.close();
}
```

Listing 10-4 Using Default Values

```
try {
    String query = "Create Table customer ( " +
                   "FirstName CHAR(30) NOT NULL," +
                   "LastName CHAR(30) NOT NULL,"+
                   "Country CHAR(30) NOT NULL DEFAULT 'USA')";
    DataRequest = Database.createStatement();
    DataRequest.executeQuery (query );
    DataRequest.close();
    Database.close();
}
```

Listing 10-5 Dropping a Table

```
try {
    String query = "Drop Table address ";
    DataRequest = Database.createStatement();
    DataRequest.executeQuery (query );
    DataRequest.close();
    Database.close();
}
```

Listing 11-1 Creating a Primary Index

```
import java.sql.*;

   public class createix
   {
      private Connection Database;

      public createix()
      {
         String url = "jdbc:odbc:customers";
```

```
        String userID = "jim";
        String password = "keogh";

        try {
               Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

               Database = DriverManager.getConnection(url,userID,password);
        }
        catch (ClassNotFoundException error) {
           System.err.println("Unable to load the JDBC/ODBC bridge." + error);
           System.exit(1);
        }
        catch (SQLException error) {
           System.err.println("Cannot connect to the database. "+ error);
           System.exit(2);
        }

        Statement DataRequest;

        try {
               String query = "CREATE UNIQUE INDEX custnum " +
                                    "ON customername (CustomerNumber) ";
               DataRequest = Database.createStatement();
               DataRequest.executeQuery (query );
               DataRequest.close();
               Database.close();

        }
        catch ( SQLException error ){
           System.err.println("SQL error." + error);
           System.exit(3);
        }

     }

  public static void main ( String args [] )
  {
     final createix link = new createix();

           System.exit ( 0 ) ;
  }
}
```

Listing 11-2 Creating a Secondary Index

```
        try {
            String query = "CREATE INDEX custlname ON customername (LastName) ";
            DataRequest = Database.createStatement();
            DataRequest.executeQuery (query );
            DataRequest.close();
            Database.close();
        }
```

Listing 11-3 Creating a Clustered Index

```
        try {
                  String query = "CREATE INDEX custLFname " +
                           " ON customername (LastName, FirstName) ";
            DataRequest = Database.createStatement();
            DataRequest.executeQuery (query);
            DataRequest.close();
            Database.close();
        }
```

Listing 11-4 Converting a Data Type

```
        try {
            String query = "CREATE INDEX custmix " +
                           "ON customername (LastName, FirstName, " +
                           " CAST CustomerNumber AS VARCHAR) ";
```

```
            DataRequest = Database.createStatement();
            DataRequest.executeQuery (query );
            DataRequest.close();
            Database.close();
        }
```

Listing 11-5 Dropping an Index

```
        try {
            String query = "DROP INDEX custlname ON customername ";
            DataRequest = Database.createStatement();
            DataRequest.executeQuery (query );
            DataRequest.close();
            Database.close();
        }
```

Listing 12-1 Inserttst

```
import java.sql.*;

   public class inserttst
   {
      private Connection Database;

      public inserttst()
      {
         String url = "jdbc:odbc:customers";
         String userID = "jim";
         String password = "keogh";

         try {
             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

             Database = DriverManager.getConnection(url,userID,password);
         }
         catch (ClassNotFoundException error) {
            System.err.println("Unable to load the JDBC/ODBC bridge." + error);
            System.exit(1);
         }
         catch (SQLException error) {
            System.err.println("Cannot connect to the database.");
            System.exit(2);
         }

         Statement DataRequest;
         try {
             String query = "INSERT INTO customername  " +
                         " VALUES ('Mary','Smith',1,'10/10/2002') ";
             DataRequest = Database.createStatement();
             DataRequest.executeQuery (query);
             DataRequest.close();
         }
         catch ( SQLException error ){
            System.err.println("SQL error." + error);
         }
      }

   public static void main ( String args [] )
   {
      final inserttst link = new inserttst();
            System.exit ( 0 ) ;
   }
}
```

Listing 12-2 Insert Data into Specific Columns

```
        try {
            String query = "INSERT INTO customername (FirstName,
            LastName,CustomerNumber,StartDate ) VALUES (
            'Bob','Jones',2,'10/11/2001') ";
```

```
        DataRequest = Database.createStatement();
        DataRequest.executeQuery (query );
        DataRequest.close();
    }
```

Listing 12-3 Making Code Easier to Read

```
    try {
        String col = "FirstName, LastName, CustomerNumber, StartDate";
        String dat = "'Bob', 'Jones', 2 , '10/11/2001'";
        String query = "INSERT INTO customername ( " + col + ") VALUES (" +
        dat +");
        DataRequest = Database.createStatement();
        DataRequest.executeQuery (query );
        DataRequest.close();
    }
```

Listing 12-4 Using CURRENT_DATE

```
    try {
        String query = "INSERT INTO customername (FirstName,
        LastName,CustomerNumber,StartDate ) VALUES (
        'Bob','Jones',2,CURRENT_DATE()) ";
        DataRequest = Database.createStatement();
        DataRequest.executeQuery (query );
        DataRequest.close();
    }
```

Listing 12-5 Using CURRENT_TIME

```
    try {
        String query = "INSERT INTO customername (FirstName,
        LastName,CustomerNumber,StartDate, StartTime ) VALUES (
        'Bob','Jones',2,CURRENT_DATE(), CURRENT_TIME()) ";
        DataRequest = Database.createStatement();
        DataRequest.executeQuery (query );
        DataRequest.close();
    }
```

Listing 12-6 Using CURRENT_TIMESTAMP

```
    try {
        String query = "INSERT INTO customername (FirstName,
        LastName,CustomerNumber,StartDateAndTime ) VALUES (
        'Bob','Jones',2,CURRENT_TIMESTAMP()) ";
        DataRequest = Database.createStatement();
        DataRequest.executeQuery (query );
        DataRequest.close();
    }
```

Listing 13-1 Selecting All the Rows and Columns

```
import java.sql.*;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import java.lang.*;

   public class selecttst extends JFrame
   {
      private JTable DataTable;
      private Connection Database;
```

```
   public selecttst()
   {
      String url = "jdbc:odbc:customers";
      String userID = "jim";
      String password = "keogh";
      Statement DataRequest;
      ResultSet Results;

      try {
          Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
          Database = DriverManager.getConnection(url,userID,password);
      }
      catch (ClassNotFoundException error) {
          System.err.println("Unable to load the JDBC/ODBC bridge." + error);
          System.exit(1);
      }
      catch (SQLException error) {
          System.err.println("Cannot connect to the database." + error);
          System.exit(2);
      }

      try {
          String query = "SELECT * FROM customers";
          DataRequest = Database.createStatement();
          Results = DataRequest.executeQuery (query);
          DisplayResults (Results);
          DataRequest.close();
      }
      catch ( SQLException error ){
          System.err.println("SQL error." + error);
          System.exit(3);
      }

      setSize (500, 200);
      show();
   }

private void DisplayResults (ResultSet DisplayResults)throws SQLException
{
   boolean Records = DisplayResults.next();

   if (!Records ) {
     JOptionPane.showMessageDialog(this, "End of data.");
   setTitle( "Process Completed");
   return;
   }
   setTitle ("Customer Names");
   Vector ColumnNames = new Vector();
   Vector rows = new Vector();

   try {
       ResultSetMetaData MetaData = DisplayResults.getMetaData();

       for ( int x = 1; x <= MetaData.getColumnCount(); ++x)
          ColumnNames.addElement (MetaData.getColumnName ( x ) );

       do {
          rows.addElement (DownRow( DisplayResults, MetaData ) );
       } while ( DisplayResults.next() );


       DataTable = new JTable (rows, ColumnNames ) ;
       JScrollPane scroller = new JScrollPane ( DataTable ) ;
       getContentPane(). add ( scroller, BorderLayout.CENTER );
       validate();
   }
   catch (SQLException error ) {
       System.err.println("Data display error." + error);
       System.exit(4);
   }
}

private Vector DownRow ( ResultSet DisplayResults, ResultSetMetaData MetaData
          )
       throws SQLException
{
   Vector currentRow = new Vector();
```

```
        for ( int x = 1; x <= MetaData.getColumnCount(); ++x )
            switch ( MetaData.getColumnType ( x ) ) {
            case Types.VARCHAR :
               currentRow.addElement( DisplayResults.getString ( x ) ) ;
               break;
            }
        return currentRow;
     }

     public static void main ( String args [] )
     {
        final selecttst link = new selecttst();

        link.addWindowListener (
             new WindowAdapter() {
            public void windowClosing(WindowEvent WinEvent )
            {
               System.exit (0) ;
            }
        }

     );
    }
  }
```

Listing 13-2 Selecting a Specific Column

```
        try {
            String query = "SELECT LastName FROM customers";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            DisplayResults (Results);
            DataRequest.close();
        }
```

Listing 13-3 Selecting More Than One Column

```
        try {
            String query = "SELECT FirstName, LastName FROM customers";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            DisplayResults (Results);
            DataRequest.close();
        }
```

Listing 13-4 Selecting Rows

```
        try {
            String query = "SELECT * " +
                           "FROM customers " +
                           "WHERE LastName = 'Jones' ";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            DisplayResults (Results);
            DataRequest.close();
        }
```

Listing 13-5 Selecting Specific Rows and Columns

```
        try {
            String query = "SELECT FirstName, LastName " +
                           "FROM customers " +
                           "WHERE LastName = 'Jones' ";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            DisplayResults (Results);
            DataRequest.close();
```

```
        }
```

Listing 13-6 Using the AND Keyword

```
        try {
            String query = "SELECT FirstName, LastName " +
                           "FROM customers " +
                           "WHERE LastName = 'Jones' " +
                           "AND FirstName = 'Bob' ";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            DisplayResults (Results);
            DataRequest.close();
        }
```

Listing 13-7 Using the OR Keyword

```
        try {
            String query = "SELECT FirstName, LastName " +
                           "FROM customers " +
                           "WHERE FirstName = 'Mary' " +
                           "OR FirstName = 'Mark' ";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            DisplayResults (Results);
            DataRequest.close();
        }
```

Listing 13-8 Using the NOT Keyword

```
        try {
            String query = "SELECT FirstName, LastName " +
                           "FROM customers " +
                           "WHERE NOT FirstName = 'Mary'";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            DisplayResults (Results);
            DataRequest.close();
        }
```

Listing 13-9 Using the Equal Operator

```
import java.sql.*;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import java.lang.*;

   public class selecttst extends JFrame
   {
      private JTable DataTable;
      private Connection Database;

      public selecttst()
      {
         String url = "jdbc:odbc:customers";
         String userID = "jim";
         String password = "keogh";
         Statement DataRequest;
         ResultSet Results;

         try {
             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

             Database = DriverManager.getConnection(url,userID,password);
         }
```

```
        catch (ClassNotFoundException error) {
            System.err.println("Unable to load the JDBC/ODBC bridge." + error);
            System.exit(1);
        }
        catch (SQLException error) {
            System.err.println("Cannot connect to the database." + error);
            System.exit(2);
        }

        try {
            String query = "SELECT * " +
                           "FROM customers " +
                           "WHERE Sales = 30000 ";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            DisplayResults (Results);
            DataRequest.close();
        }
        catch ( SQLException error ){
            System.err.println("SQL error.");
            System.exit(3);
        }

        setSize (500, 200);
        show();
    }

private void DisplayResults (ResultSet DisplayResults)throws SQLException
{
    boolean Records = DisplayResults.next();

    if (!Records ) {
        JOptionPane.showMessageDialog( this, "End of data.");
    setTitle( "Process Completed");
    return;
    }
    setTitle ("Customer Names");
    Vector ColumnNames = new Vector();
    Vector rows = new Vector();

    try {
        ResultSetMetaData MetaData = DisplayResults.getMetaData();

        for ( int x = 1; x <= MetaData.getColumnCount(); ++x)
            ColumnNames.addElement (MetaData.getColumnName ( x ) );

        do {
            rows.addElement (DownRow( DisplayResults, MetaData ) );
        } while ( DisplayResults.next() );


        DataTable = new JTable ( rows, ColumnNames ) ;
        JScrollPane scroller = new JScrollPane ( DataTable ) ;
        getContentPane(). add ( scroller, BorderLayout.CENTER );
        validate();
    }
    catch (SQLException error ) {
        System.err.println("Data display error." + error);
        System.exit(4);
    }
}

private Vector DownRow ( ResultSet DisplayResults, ResultSetMetaData MetaData
          )
        throws SQLException
{
    Vector currentRow = new Vector();
    for ( int x = 1; x <= MetaData.getColumnCount(); ++x )
        switch ( MetaData.getColumnType ( x ) ) {
        case Types.VARCHAR :
            currentRow.addElement( DisplayResults.getString ( x ) ) ;
            break;
        case Types.INTEGER :
            currentRow.addElement( new Long( DisplayResults.getLong( x )) ) ;
            break;
        }
    return currentRow;
```

```
   }

   public static void main ( String args [] )
   {
      final selecttst link = new selecttst();

      link.addWindowListener (
                 new WindowAdapter() {
          public void windowClosing(WindowEvent WinEvent )
          {

             System.exit ( 0 ) ;
          }
      }

   );
  }
}
```

Listing 13-10 Using the Not Equal Operator

```
try {
    String query = "SELECT * " +
                   "FROM customers " +
                   "WHERE Sales <> 30000 ";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 13-11 Using the Less Than Operator

```
try {
    String query = "SELECT * " +
                   "FROM customers " +
                   "WHERE Sales < 30000 ";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 13-12 Using the Greater Than Operator

```
try {
    String query = "SELECT * " +
                   "FROM customers " +
                   "WHERE Sales > 30000 ";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 13-13 Using the Less Than Equal Operator

```
try {
    String query = "SELECT * " +
                   "FROM customers " +
                   "WHERE Sales <= 30000 ";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 13-14 Using the Greater Than Equal Operator

```
try {
    String query = "SELECT * " +
                   "FROM customers " +
                   "WHERE Sales >= 30000 ";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 13-15 Using the BETWEEN Operator

```
try {
    String query = "SELECT * " +
                   "FROM customers " +
                   "WHERE Sales BETWEEN 20000 AND 39999";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 13-16 Using the LIKE Operator

```
try {
    String query = "SELECT * " +
                   "FROM customers " +
                   "WHERE LastName LIKE 'Smi%'";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 13-17 Using IS NULL

```
try {
    String query = "SELECT * " +
                   "FROM customers " +
                   "WHERE State IS NULL";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 13-18 Using the DISTINCT Keyword

```
try {
    String query = "SELECT DISTINCT * " +
                   "FROM customers ";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 13-19 Using the IN Keyword

```
try {
    String query = "SELECT * " +
```

```
                    "FROM customers " +
                    "WHERE Sales IN (10000, 20000, 30000)";
        DataRequest = Database.createStatement();
        Results = DataRequest.executeQuery (query);
        DisplayResults (Results);
        DataRequest.close();
    }
```

Listing 13-20 Using the NOT IN Keywords

```
    try {
        String query = "SELECT * " +
                    "FROM customers " +
                    "WHERE Sales NOT IN (10000, 20000, 30000)";
        DataRequest = Database.createStatement();
        Results = DataRequest.executeQuery (query);
        DisplayResults (Results);
        DataRequest.close();
    }
```

Listing 14-1 Getting Return Data from the Console Window

```
import java.sql.*;

   public class plainselect
   {
      private Connection Database;

      public plainselect()
      {
         String url = "jdbc:odbc:customers";
         String userID = "jim";
         String password = "keogh";
         Statement DataRequest;
         ResultSet Results;

         try {
             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

             Database = DriverManager.getConnection(url,userID,password);
         }
         catch (ClassNotFoundException error) {
             System.err.println("Unable to load the JDBC/ODBC bridge." + error);
             System.exit(1);
         }
         catch (SQLException error) {
             System.err.println("Cannot connect to the database." + error);
             System.exit(2);
         }

         try {
             String query = "Select DISTINCT * " +
                            "FROM customers " +
                            "WHERE Sales IN (10000, 20000, 30000)";
             DataRequest = Database.createStatement();
             Results = DataRequest.executeQuery (query);
             DisplayResults (Results);
             DataRequest.close();
         }
         catch ( SQLException error ){
             System.err.println("SQL error.");
             System.exit(3);
         }
         shutDown();
      }

   private void DisplayResults (ResultSet DisplayResults)throws SQLException
   {

      boolean Records = DisplayResults.next();

      if (!Records ) {
```

```
      System.out.println( "End of data.");
   return;
   }

   try {
       do {
          DownRow( DisplayResults) ;
       } while ( DisplayResults.next() );


   }
   catch (SQLException error ) {
       System.err.println("Data display error." + error);
       System.exit(4);
   }
}

private  void DownRow ( ResultSet DisplayResults )
       throws SQLException
{
   String FirstName= new String();
   String LastName= new String();
   String Street= new String();
   String City = new String();
   String State = new String();
   String ZipCode= new String();
   long Sales;
   long Profit;
   String printrow;

   FirstName = DisplayResults.getString ( 1 ) ;
   LastName = DisplayResults.getString ( 2 ) ;
   Street = DisplayResults.getString ( 3 ) ;
   City = DisplayResults.getString ( 4 ) ;
   State = DisplayResults.getString ( 5 ) ;
   ZipCode = DisplayResults.getString ( 6 ) ;
   Sales = DisplayResults.getLong ( 7 ) ;
   Profit = DisplayResults.getLong ( 8 ) ;
   printrow = FirstName + " " +
            LastName + " " +
            City + " " +
                  State +  " " +
            ZipCode + " " +
            Sales + " " +
            Profit;
   System.out.println(printrow);
}

public void shutDown()
{
   try {
       Database.close();
   }
   catch (SQLException error) {
      System.err.println( "Cannot break connection." + error ) ;
   }
}
public static void main ( String args [] )
{
   final plainselect link = new plainselect();
           System.exit ( 0 ) ;
}
}
```

Listing 14-2 Using Other Types of Data

```
private  void DownRow ( ResultSet DisplayResults )
      throws SQLException
{
   String FirstName= new String();
   String LastName= new String();
   String Street= new String();
   String City = new String();
   String State = new String();
   String ZipCode= new String();
```

```
    java.sql.Date StartDate ;
    java.sql.Time StartTime;
    double Sales;
    double Profit;
    boolean NewCustomer;
    String printrow;

    FirstName = DisplayResults.getString ( 1 ) ;
    LastName = DisplayResults.getString ( 2 ) ;
    Street = DisplayResults.getString ( 3 ) ;
    City = DisplayResults.getString ( 4 ) ;
    State = DisplayResults.getString ( 5 ) ;
    ZipCode = DisplayResults.getString ( 6 ) ;
    Sales = DisplayResults.getDouble ( 7 ) ;
    Profit = DisplayResults.getDouble ( 8 ) ;
    StartDate = DisplayResults.getDate ( 9 ) ;
    StartTime = DisplayResults.getTime ( 10 ) ;
    NewCustomer = DisplayResults.getBoolean ( 11 ) ;
    printrow = FirstName + " " +
             LastName + " " +
             City + " " +
             State +  " " +
             ZipCode + " " +
             Sales + " " +
             Profit + " " +
             StartDate + " " +
             StartTime + " " +
             NewCustomer;
    System.out.println(printrow);
}
```

Listing 14-3 Reading the Number of Columns

```
private  void DownRow ( ResultSet DisplayResults )
      throws SQLException
{
   ResultSetMetaData metadata = DisplayResults.getMetaData ();
   int NumberOfColumns;
   String printrow;

   NumberOfColumns = metadata.getColumnCount () ;

  System.out.println("Number Of Columns: " + NumberOfColumns);
}
```

Listing 14-4 Determining the Data Type of a Column

```
private  void DownRow ( ResultSet DisplayResults )
      throws SQLException
{
   ResultSetMetaData metadata = DisplayResults.getMetaData ();
   String ColumnType = new String();
   String printrow;

   ColumnType = metadata.getColumnTypeName ( 9 ) ;

  System.out.println("Column Type: " + ColumnType );
}
```

Listing 14-5 Copying the Column Name from the Metadata

```
private  void DownRow ( ResultSet DisplayResults )
      throws SQLException
{
   ResultSetMetaData metadata = DisplayResults.getMetaData ();
   String ColumnName = new String();
   String printrow;

   ColumnName = metadata.getColumnLabel (9) ;
```

```
   System.out.println("Column Name: " + ColumnName);
 }
```

Listing 14-6 Determining the Column Size

```
 private  void DownRow ( ResultSet DisplayResults )
       throws SQLException
 {
    ResultSetMetaData metadata = DisplayResults.getMetaData ();
    int ColumnWidth;
    String printrow;

    ColumnWidth = metadata.getColumnDisplaySize ( 9 ) ;

   System.out.println("Column Width:" + ColumnWidth);
 }
```

Listing 14-7 Displaying Data in a Table Format

```
import java.sql.*;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import java.lang.*;

   public class tabletst extends JFrame
   {
      private JTable DataTable;
      private Connection Database;

      public tabletst ()
      {
         String url = "jdbc:odbc:customers";
         String userID = "jim";
         String password = "keogh";
         Statement DataRequest;
         ResultSet Results;

         try {
             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
             Database = DriverManager.getConnection(url,userID,password);
         }
         catch (ClassNotFoundException error) {
             System.err.println("Unable to load the JDBC/ODBC bridge." + error);
             System.exit(1);
         }
         catch (SQLException error) {
             System.err.println("Cannot connect to the database." + error);
             System.exit(2);
         }

         try {
             String query = "SELECT FirstName, LastName, Street, City FROM
             customers";
             DataRequest = Database.createStatement();
             Results = DataRequest.executeQuery (query);
             DisplayResults (Results);
             DataRequest.close();
         }
         catch ( SQLException error ){
             System.err.println("SQL error." + error);
             System.exit(3);
         }

         setSize (500, 200);
         show();
      }

   private void DisplayResults (ResultSet DisplayResults)throws SQLException
   {
      boolean Records = DisplayResults.next();
```

```
        if (!Records ) {
           JOptionPane.showMessageDialog(this, "End of data.");
        setTitle( "Process Completed");
        return;
        }
        setTitle ("Customer Names");
        Vector ColumnNames = new Vector();
        Vector rows = new Vector();

        try {
            ResultSetMetaData MetaData = DisplayResults.getMetaData();

            for ( int x = 1; x <= MetaData.getColumnCount(); ++x)
               ColumnNames.addElement (MetaData.getColumnName ( x ) );

            do {
               rows.addElement (DownRow( DisplayResults, MetaData ) );
            } while ( DisplayResults.next() );


            DataTable = new JTable (rows, ColumnNames ) ;
            JScrollPane scroller = new JScrollPane ( DataTable ) ;
            getContentPane(). add ( scroller, BorderLayout.CENTER );
            validate();
        }
        catch (SQLException error ) {
            System.err.println("Data display error." + error);
            System.exit(4);
        }
     }

     private Vector DownRow ( ResultSet DisplayResults, ResultSetMetaData MetaData
               )
            throws SQLException
     {
        Vector currentRow = new Vector();
        for ( int x = 1; x <= MetaData.getColumnCount(); ++x )
            switch ( MetaData.getColumnType ( x ) ) {
            case Types.VARCHAR :
               currentRow.addElement( DisplayResults.getString ( x ) ) ;
               break;
            }
        return currentRow;
     }

     public static void main ( String args [] )
     {
        final tabletst link = new tabletst ();

        link.addWindowListener (
             new WindowAdapter() {
            public void windowClosing(WindowEvent WinEvent )
            {
              System.exit (0) ;
            }
        }

     );
    }
  }
```

Listing 14-8 Displaying a GUI Form

```
import java.sql.*;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import java.lang.*;

   public class guiform extends JFrame
   {
      private Connection Database;
      public static final int WIDTH = 300;
```

```
   public static final int HEIGHT = 200;

   public guiform()
   {
      String url = "jdbc:odbc:customers";
      String userID = "jim";
      String password = "keogh";
      Statement DataRequest;
      ResultSet Results;

      try {
          Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

          Database = DriverManager.getConnection(url,userID,password);
      }
      catch (ClassNotFoundException error) {
          System.err.println("Unable to load the JDBC/ODBC bridge." + error);
          System.exit(1);
      }
      catch (SQLException error) {
          System.err.println("Cannot connect to the database." + error);
          System.exit(2);
      }

      try {
          String query = "SELECT DISTINCT FirstName, LastName " +
                         "FROM customers " +
                         "WHERE FirstName = 'Mary'" +
                         "AND LastName = 'Smith'";
          DataRequest = Database.createStatement();
          Results = DataRequest.executeQuery (query);
          DisplayResults (Results);
          DataRequest.close();
      }
      catch ( SQLException error ){
          System.err.println("SQL error.");
          System.exit(3);
      }
   }

private void DisplayResults (ResultSet DisplayResults)throws SQLException
{
   try{
      boolean Records = DisplayResults.next();

      if (!Records ) {
           JOptionPane.showMessageDialog( this, "End of data.");
           setTitle( "Process Completed");
      return;
      }

      String FirstName= new String();
      String LastName= new String();

      FirstName = DisplayResults.getString ( 1 ) ;
      LastName = DisplayResults.getString ( 2 ) ;

      setTitle("Customer");
      setSize(WIDTH,HEIGHT);
      JLabel label1= new JLabel("First Name:");
      JTextField textField1 = new JTextField(10);
      textField1.setMaximumSize(textField1.getPreferredSize());
      textField1.setText(FirstName);

      Box hbox1 = Box.createHorizontalBox();
      hbox1.add(label1);
      hbox1.add(Box.createHorizontalStrut(10));
      hbox1.add(textField1);

      JLabel label2 = new JLabel("Last Name:");
      JTextField textField2 = new JTextField(10);
      textField2.setMaximumSize(textField2.getPreferredSize());
      textField2.setText(LastName);

      Box hbox2 = Box.createHorizontalBox();
      hbox2.add(label2);
      hbox2.add(Box.createHorizontalStrut(10));
```

```
         hbox2.add(textField2);

         Box vbox = Box.createVerticalBox();
         vbox.add(hbox1);
         vbox.add(hbox2);
         vbox.add(Box.createGlue());

         Container contentPane = getContentPane();
         contentPane.add(vbox, BorderLayout.CENTER);
      }

      catch ( SQLException error ){
         System.err.println("SQL error." + error);
         System.exit(3);
      }
   }

   public static void main ( String args [] )
   {
      final guiform link = new guiform();

      link.addWindowListener(new
         WindowAdapter()
            {
            public void windowClosing(WindowEvent e)
               {
                  System.exit(0);
               }
            }
         );
      link.show();

      }
   }
```

Listing 15-1 Updating Data in a Table

```
import java.sql.*;

   public class updatetst
   {
      private Connection Database;

      public updatetst()
      {
         String url = "jdbc:odbc:customers";
         String userID = "jim";
         String password = "keogh";
         Statement DataRequest;
         ResultSet Results;

         try {
             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

             Database = DriverManager.getConnection(url,userID,password);
         }
         catch (ClassNotFoundException error) {
             System.err.println("Unable to load the JDBC/ODBC bridge." + error);
             System.exit(1);
         }
         catch (SQLException error) {
             System.err.println("Cannot connect to the database." + error);
             System.exit(2);
         }

         try {
             String query = "UPDATE customers " +
                            "SET Street = '5 Main Street' " +
                            "WHERE FirstName = 'Bob'";
             DataRequest = Database.createStatement();
             Results = DataRequest.executeQuery (query );
             DataRequest.close();
         }
         catch ( SQLException error ){
             System.err.println("SQL error." + error);
```

```
            System.exit(3);
        }
    }

  public static void main ( String args [] )
  {
    final updatetst link = new updatetst();
    System.exit ( 0 ) ;
  }
}
```

Listing 15-2 Using the IN Test

```
try {
    String query = "UPDATE customers " +
                   "SET Discount = 25 " +
                   "WHERE Discount IN (12,15)";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DataRequest.close();
}
```

Listing 15-3 Using the IS NULL Test

```
try {
    String query = "UPDATE customers " +
                   "SET Discount = 0 " +
                   "WHERE LastName IS NULL  ";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DataRequest.close();
}
```

Listing 15-4 Using a Comparison Operator

```
try {
    String query = "UPDATE customers " +
                   "SET Discount = 20 " +
                   "WHERE Discount > 20 ";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DataRequest.close();
}
```

Listing 15-5 Updating All Rows

```
try {
    String query = "UPDATE customers " +
                   "SET Discount = 0 ";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DataRequest.close();
}
```

Listing 15-6 Updating Multiple Columns at One Time

```
try {
    String query = "UPDATE customers " +
                   "SET Discount = 12, " +
                   "Street = 'Jones Street' " +
                   "WHERE LastName = 'Jones'";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DataRequest.close();
```

```
        }
```

Listing 15-7 Making a Calculation

```
        try {
            String query = "UPDATE customers " +
                           "SET DiscountPrice = " +
                           "Price * ((100 - Discount) / 100) ";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            DataRequest.close();
        }
```

Listing 16-1 Deleting a Row of Data

```
import java.sql.*;

   public class deletetst
   {
      private Connection Database;

      public deletetst()
      {
         String url = "jdbc:odbc:customers";
         String userID = "jim";
         String password = "keogh";
         Statement DataRequest;
         ResultSet Results;

         try {
             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

             Database = DriverManager.getConnection(url,userID,password);
         }
         catch (ClassNotFoundException error) {
             System.err.println("Unable to load the JDBC/ODBC bridge." + error);
             System.exit(1);
         }
         catch (SQLException error) {
             System.err.println("Cannot connect to the database." + error);
             System.exit(2);
         }

         try {
             String query = "DELETE FROM customers " +
                            "WHERE LastName = 'Jones' and FirstName = 'Tom'";
             DataRequest = Database.createStatement();
             Results = DataRequest.executeQuery (query );
             DataRequest.close();
         }
         catch ( SQLException error ){
             System.err.println("SQL error." + error);
             System.exit(3);
         }
      }

   public static void main ( String args [] )
   {
      final deletetst link = new deletetst();
         System.exit ( 0 ) ;
    }
}
```

Listing 16-2 Using the IN Test to Delete Rows

```
        try {
            String query = "DELETE FROM customers " +
                           "WHERE Discount IN (3,5)";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query );
```

```
        DataRequest.close();
    }
```

Listing 16-3 Using IS NULL to Delete Rows

```
    try {
        String query = "DELETE FROM customers " +
                       "WHERE FirstName IS NULL  ";
        DataRequest = Database.createStatement();
        Results = DataRequest.executeQuery (query );
        DataRequest.close();
    }
```

Listing 16-4 Using a Comparison Operator to Delete Rows

```
    try {
        String query = "DELETE FROM customers " +
                       "WHERE Discount > 20 ";
        DataRequest = Database.createStatement();
        Results = DataRequest.executeQuery (query );
        DataRequest.close();
    }
```

Listing 16-5 Deleting All Rows

```
    try {
        String query = "DELETE FROM customers ";
        DataRequest = Database.createStatement();
        Results = DataRequest.executeQuery (query);
        DataRequest.close();
    }
```

Listing 17-1 Creating an Equal Join

```
import java.sql.*;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import java.lang.*;

   public class jointst extends JFrame
   {
      private JTable DataTable;
      private Connection Database;

      public jointst()
      {
         String url = "jdbc:odbc:customers";
         String userID = "jim";
         String password = "keogh";
         Statement DataRequest;
         ResultSet Results;

         try {
             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

             Database = DriverManager.getConnection(url,userID,password);
         }
         catch (ClassNotFoundException error) {
             System.err.println("Unable to load the JDBC/ODBC bridge."  + error);
             System.exit(1);
         }
         catch (SQLException error) {
             System.err.println("Cannot connect to the database."  + error);
             System.exit(2);
         }
```

```
      try {
          String query = " SELECT FirstName,LastName,State,ZipCode " +
                         "FROM Customers, ZipCode " +
                         "WHERE Zip = ZipCode";
          DataRequest = Database.createStatement();
          Results = DataRequest.executeQuery (query);
          DisplayResults (Results);
          DataRequest.close();
      }
      catch ( SQLException error ){
          System.err.println("SQL error." + error);
          System.exit(3);
      }

      setSize (500, 200);
      show();
   }

private void DisplayResults (ResultSet DisplayResults)throws SQLException
{
   boolean Records = DisplayResults.next();

   if (!Records ) {
      JOptionPane.showMessageDialog( this, "End of data.");
   setTitle( "Process Completed");
   return;
   }
   setTitle ("Customer Names");
   Vector ColumnNames = new Vector();
   Vector rows = new Vector();

   try {
       ResultSetMetaData MetaData = DisplayResults.getMetaData();

       for ( int x = 1; x <= MetaData.getColumnCount(); ++x)
          ColumnNames.addElement (MetaData.getColumnName ( x ) );

       do {
          rows.addElement (DownRow( DisplayResults, MetaData ) );
       } while ( DisplayResults.next() );


       DataTable = new JTable (rows, ColumnNames ) ;
       JScrollPane scroller = new JScrollPane (DataTable ) ;
       getContentPane(). add ( scroller, BorderLayout.CENTER );
       validate();
   }
   catch (SQLException error ) {
       System.err.println("Data display error."  + error);
       System.exit(4);
   }
}

private Vector DownRow ( ResultSet DisplayResults, ResultSetMetaData MetaData
          )
       throws SQLException
{
   Vector currentRow = new Vector();
   for ( int x = 1; x <= MetaData.getColumnCount(); ++x )
       switch ( MetaData.getColumnType ( x ) ) {
       case Types.VARCHAR :
          currentRow.addElement( DisplayResults.getString ( x ) ) ;
          break;
       case Types.INTEGER :
          currentRow.addElement( new Long( DisplayResults.getLong( x )) ) ;
          break;
       }
   return currentRow;
}

public static void main ( String args [] )
{
   final jointst link = new jointst();

   link.addWindowListener (
              new WindowAdapter() {
```

```
            public void windowClosing(WindowEvent WinEvent )
            {

               System.exit ( 0 ) ;
            }
        }

      );
    }
}
```

Listing 17-2 Using a Parent-Child Join

```
        try {
            String query = " SELECT OrderNumber, ProductName " +
                           " FROM Orders, Products " +
                           " WHERE ProdNumber = ProductNumber";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            DisplayResults (Results);
            DataRequest.close();
        }
```

Listing 17-3 Using Several Comparison Joins

```
        try {
            String query = " SELECT OrderNumber, ProductName, Quantity " +
                           " FROM Orders, Products " +
                           " WHERE ProdNumber = ProductNumber " +
                           " AND Quantity > 2";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            DisplayResults (Results);
            DataRequest.close();
        }
```

Listing 17-4 Joining Three or More Tables

```
  try {
      String query = " SELECT FirstName, LastName,OrderNumber, ProductName,
            Quantity " +
                     " FROM customers, Orders, Products " +
                     " WHERE ProdNumber = ProductNumber " +
                           " AND CustNumber = CustomerNumber";
           DataRequest = Database.createStatement();
           Results = DataRequest.executeQuery (query);
           DisplayResults (Results);
           DataRequest.close();
        }
```

Listing 17-5 Using a Column Name Qualifier

```
  try {
      String query = "SELECT customers.CustomerNumber , " +
                           " FirstName, LastName, OrderNumber, " +
                           " ProductName, Quantity " +
                     " FROM customers, Orders, Products " +
                     " WHERE ProdNumber = ProductNumber " +
                           " AND customers.CustomerNumber =
            Orders.CustomerNumber";
        DataRequest = Database.createStatement();
        Results = DataRequest.executeQuery (query);
        DisplayResults (Results);
        DataRequest.close();
     }
```

Listing 17-6 Using a Table Alias

```
try {
    String query = "SELECT c.CustomerNumber , " +
                     " c.FirstName, c.LastName, o.OrderNumber, " +
                     " p.ProductName, o.Quantity " +
                  " FROM customers c, Orders o, Products  p" +
                  " WHERE o.ProdNumber = p.ProductNumber " +
                        " AND c.CustomerNumber = o.CustomerNumber";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 17-7 Creating an Inner Join

```
try {
    String query = " SELECT FirstName, LastName,OrderNumber, ProductName,
          Quantity " +
                   " FROM Customers,Orders, Products " +
                   " WHERE ProdNumber = ProductNumber " +
                           " AND Customers.CustomerNumber =
          Orders.CustomerNumber";
      DataRequest = Database.createStatement();
      Results = DataRequest.executeQuery (query);
      DisplayResults (Results);
      DataRequest.close();
    }
```

Listing 17-8 Creating a Left Outer Join

```
try {
    String query = " SELECT FirstName, LastName,OrderNumber" +
                   " FROM Customers c, Orders o" +
                   " WHERE c.CustomerNumber *= o.CustomerNumber";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 17-9 Creating a Left Outer Join in Microsoft Access

```
try {
    String query = " SELECT FirstName, LastName,OrderNumber " +
                   "  FROM Customers LEFT JOIN Orders " +
                      "  ON Customers.CustomerNumber =
    Orders.CustomerNumber";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 17-10 Creating a Right Outer Join

```
try {
    String query = " SELECT FirstName, LastName,OrderNumber" +
                   " FROM Customers c, Orders o" +
                   " WHERE c.CustomerNumber =* o.CustomerNumber";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DisplayResults (Results);
    DataRequest.close();
```

```
        }
```

Listing 17-11 Creating a Right Join in Microsoft Access

```
        try {
            String query = "SELECT FirstName, LastName,OrderNumber " +
                           "  FROM Customers RIGHT JOIN Orders " +
                           "  ON Customers.CustomerNumber =
            Orders.CustomerNumber";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            DisplayResults (Results);
            DataRequest.close();
        }
```

Listing 17-12 Creating a Full Outer Join

```
        try {
            String query = "SELECT FirstName, LastName,OrderNumber " +
                           " FROM Customers c, Orders o" +
                           " WHERE c.CustomerNumber *=* o.CustomerNumber";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            DisplayResults (Results);
            DataRequest.close();
        }
```

Listing 18-1 Grouping Rows by the Value in One Column

```
import java.sql.*;
public class grouptst
   {
      private Connection Database;

      public grouptst ()
      {
         String url = "jdbc:odbc:customers";
         String userID = "jim";
         String password = "keogh";
         Statement DataRequest;
         ResultSet Results;

         try {
             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

             Database = DriverManager.getConnection(url,userID,password);
         }
         catch (ClassNotFoundException error) {
             System.err.println("Unable to load the JDBC/ODBC bridge." + error);
             System.exit(1);
         }
         catch (SQLException error) {
             System.err.println("Cannot connect to the database." + error);
             System.exit(2);
         }

         try {
             String query = " SELECT Store, SUM(Sales) " +
                            " FROM sales " +
                            " GROUP BY Store";
             DataRequest = Database.createStatement();
             Results = DataRequest.executeQuery (query);
             System.out.println("Store     Sales");
             System.out.println("-----     -----");
             DisplayResults (Results);
             DataRequest.close();
         }
         catch ( SQLException error ){
             System.err.println("SQL error." + error);
             System.exit(3);
```

```
        }
        shutDown();
     }

   private void DisplayResults (ResultSet DisplayResults)throws SQLException
   {

      boolean Records = DisplayResults.next();

      if (!Records ) {
        System.out.println( "End of data.");
      return;
      }

      try {
          do {
             DownRow( DisplayResults) ;
          } while ( DisplayResults.next() );

      }
      catch (SQLException error ) {
          System.err.println("Data display error." + error);
          System.exit(4);
      }
   }

   private  void DownRow ( ResultSet DisplayResults )
          throws SQLException
   {
      String Store;
      long Sum;
      Store = DisplayResults.getString ( 1 ) ;
      Sum = DisplayResults.getLong ( 2 ) ;
      System.out.println(Store + "         " + Sum);
   }

   public void shutDown()
   {
      try {
          Database.close();
      }
      catch (SQLException error) {
          System.err.println( "Cannot break connection." + error ) ;
      }
   }
   public static void main ( String args [] )
   {
      final grouptst link = new grouptst ();
              System.exit ( 0 ) ;
   }
}
```

Listing 18-2 Grouping Multiple Columns

```
         try {
             String query = " SELECT Store,SalesRep, SUM(Sales) " +
                            " FROM sales " +
                            " GROUP BY Store, SalesRep";
             DataRequest = Database.createStatement();
             Results = DataRequest.executeQuery (query);
             System.out.println("Store  SalesRep   Sales");
             System.out.println("-----  --------   -----");
             DisplayResults (Results);
             DataRequest.close();
         }
```

Listing 18-3 Replacing the DownRow() Method

```
   private  void DownRow ( ResultSet DisplayResults )
         throws SQLException
   {
      String Store;
```

```
    String SalesRep;
    long Sum;
    Store = DisplayResults.getString ( 1 ) ;
    SalesRep = DisplayResults.getString ( 2 ) ;
    Sum = DisplayResults.getLong ( 3 ) ;
    System.out.println(Store + "       " + SalesRep + "       " + Sum);
 }
```

Listing 18-4 Using the HAVING Clause

```
       try {
           String query = "SELECT Store, SUM(Sales) " +
                          " FROM sales " +
                          " GROUP BY Store" +
                          " HAVING SUM(Sales) > 400";
           DataRequest = Database.createStatement();
           Results = DataRequest.executeQuery (query);
           System.out.println("Store     Sales");
           System.out.println("-----     -----");
           DisplayResults (Results);
           DataRequest.close();
       }
```

Listing 18-5 Displaying Store and Sales Columns Onscreen

```
 private  void DownRow ( ResultSet DisplayResults )
       throws SQLException
 {
    String Store;
    long Sum;
    Store = DisplayResults.getString ( 1 ) ;
    Sum = DisplayResults.getLong ( 2 ) ;
    System.out.println(Store + "         " + Sum);
 }
```

Listing 18-6 Sorting Values

```
       try {
           String query = "SELECT Store, Sales " +
                          " FROM sales " +
                          " ORDER BY Store";
           DataRequest = Database.createStatement();
           Results = DataRequest.executeQuery (query);
           System.out.println("Store     Sales");
           System.out.println("-----     -----");
           DisplayResults (Results);
           DataRequest.close();
       }
```

Listing 18-7 Replace DownRow() to Display Two Columns

```
 private  void DownRow ( ResultSet DisplayResults )
       throws SQLException
 {
    String Store;
    long Sales;
    Store = DisplayResults.getString ( 1 ) ;
    Sales = DisplayResults.getLong ( 2 ) ;
    System.out.println(Store + "         " + Sales);
 }
```

Listing 18-8 Sorting with Major and Minor Keys

```
       try {
```

```
    String query = "SELECT  Store, Sales " +
                   " FROM sales " +
                   " ORDER BY Store, Sales";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    System.out.println("Store     Sales");
    System.out.println("-----     -----");
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 18-9 Using a Descending Sort

```
try {
    String query = "SELECT  Store, Sales " +
                   " FROM sales " +
                   " ORDER BY Store DESC";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    System.out.println("Store     Sales");
    System.out.println("-----     -----");
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 18-10 Using an Ascending Sort

```
try {
    String query = "SELECT  Store, Sales " +
                   " FROM sales " +
                   " ORDER BY Store ASC";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    System.out.println("Store     Sales");
    System.out.println("-----     -----");
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 18-11 Using Both Ascending and Descending Sorts

```
try {
    String query = "SELECT  Store, Sales " +
                   " FROM sales " +
                   " ORDER BY Store DESC, Sales ASC";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    System.out.println("Store     Sales");
    System.out.println("-----     -----");
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 18-12 Sorting a Column in a ResultSet

```
try {
    String query = "SELECT Store, (Sales-Estimate) " +
                   " FROM sales " +
                   " ORDER BY 2 ";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    System.out.println("Store     Difference");
    System.out.println("-----     -----");
    DisplayResults (Results);
    DataRequest.close();
}
```

Listing 19-1 Using a Subquery

```
import java.sql.*;

   public class squtst
   {
      private Connection Database;

      public squtst()
      {
         String url = "jdbc:odbc:customers";
         String userID = "jim";
         String password = "keogh";
         Statement DataRequest;
         ResultSet Results;

         try {
             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

             Database = DriverManager.getConnection(url,userID,password);
         }
         catch (ClassNotFoundException error) {
             System.err.println("Unable to load the JDBC/ODBC bridge." + error);
             System.exit(1);
         }
         catch (SQLException error) {
             System.err.println("Cannot connect to the database." + error);
             System.exit(2);
         }

         try {
             String query = " SELECT Store " +
                            " FROM sales "+
                            " WHERE Estimate = " +
                               " (SELECT SUM(Amount) " +
                                       " FROM Orders " +
                                       " WHERE StoreNum = Store) ";
             DataRequest = Database.createStatement();
             Results = DataRequest.executeQuery (query);
             System.out.println("Store");
             System.out.println("-----");
             DisplayResults (Results);
             DataRequest.close();
         }
         catch ( SQLException error ){
             System.err.println("SQL error." + error);
             System.exit(3);
         }
         shutDown();
      }

   private void DisplayResults (ResultSet DisplayResults)throws SQLException
   {

      boolean Records = DisplayResults.next();

      if (!Records ) {
        System.out.println( "End of data.");
      return;
      }

      try {
          do {
             DownRow( DisplayResults) ;
          } while ( DisplayResults.next() );

      }
      catch (SQLException error ) {
          System.err.println("Data display error." + error);
          System.exit(4);
      }
   }

   private  void DownRow ( ResultSet DisplayResults )
          throws SQLException
```

```
    {
       String Store;
       Store = DisplayResults.getString ( 1 ) ;
       System.out.println(Store);
    }

    public void shutDown()
    {
       try {
           Database.close();
       }
       catch (SQLException error) {
           System.err.println( "Cannot break connection." + error ) ;
       }
    }
    public static void main ( String args [] )
    {
       final squtst link = new squtst();
              System.exit ( 0 ) ;
    }
}
```

Listing 19-2 Using Comparison Operators

```
        try {
            String query = " SELECT Store " +
                           " FROM sales "+
                           " WHERE Estimate < " +
                               " (SELECT SUM(Amount) " +
                                       " FROM Orders " +
                                       " WHERE StoreNum = Store) ";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            System.out.println("Store");
            System.out.println("-----");
            DisplayResults (Results);
            DataRequest.close();
        }
```

Listing 19-3 Using the HAVING Clause

```
        try {
            String query = " SELECT Store " +
                           " FROM sales  "+
                           " GROUP BY store " +
                           " HAVING SUM(Estimate) < " +
                                " (SELECT SUM(Amount) " +
                                " FROM Orders " +
                                " WHERE StoreNum = Store) ";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            System.out.println("Store");
            System.out.println("-----");
            DisplayResults (Results);
            DataRequest.close();
        }
```

Listing 19-4 Using the Existence Test

```
        try {
            String query = " SELECT DISTINCT Store " +
                           " FROM sales  "+
                           " WHERE EXISTS " +
                                " (SELECT StoreNum " +
                                            " FROM Orders " +
                                     " WHERE StoreNum = Store) ";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query);
            System.out.println("Store");
            System.out.println("-----");
```

```
        DisplayResults (Results);
        DataRequest.close();
    }
```

Listing 19-5 Using the Set Membership Test

```
    try {
        String query = " SELECT SalesRep " +
                              " FROM sales  "+
                              " WHERE Store IN " +
                                 " (SELECT StoreNum " +
                                         " FROM Orders " +
                                   " WHERE Estimate < Amount) ";
        DataRequest = Database.createStatement();
        Results = DataRequest.executeQuery (query);
        System.out.println("Store");
        System.out.println("-----");
        DisplayResults (Results);
        DataRequest.close();
    }
```

Listing 19-6 Using the ANY Qualified Comparison Test

```
    try {
        String query = " SELECT DISTINCT Store " +
                       " FROM sales  "+
                       " WHERE Estimate  > ANY  " +
                          " (SELECT Amount" +
                                " FROM Orders " +
                           " WHERE store = StoreNum) ";
        DataRequest = Database.createStatement();
        Results = DataRequest.executeQuery (query);
        System.out.println("Store");
        System.out.println("-----");
        DisplayResults (Results);
        DataRequest.close();
    }
```

Listing 19-7 Using the ALL Qualified Comparison Test

```
    try {
        String query = " SELECT DISTINCT Store " +
                       " FROM sales  "+
                       " WHERE Estimate  > ALL  " +
                          " (SELECT Amount" +
                                " FROM Orders " +
                           " WHERE store = StoreNum) ";
        DataRequest = Database.createStatement();
        Results = DataRequest.executeQuery (query);
        System.out.println("Store");
        System.out.println("-----");
        DisplayResults (Results);
        DataRequest.close();
    }
```

Bonus Chapter 1

You've Got That Look: Views

In This Chapter

- Creating a view
- Using a view
- Creating a horizontal view
- Creating a vertical view

Report-card time is always humorous in my house because of the way my daughter shows us her report card. She uses her hands to block rows of subjects and columns of grades that she doesn't want us to see.

You may need to do something similar with your database. Instead of hiding data with your hands, you can hide data by creating a view of the database. Sometimes you don't want other applications and applets to have access to certain groups of rows or columns.

A *view* is a virtual table that's created using columns and rows of one or more tables in a database. A view

- Gives the appearance of a table but isn't a real table.
- Doesn't have its own rows and columns. Instead, it borrows rows and columns from other tables.
- Can be used in queries just like you'd use a real table.
- Restricts access to data in a database.
- Gives each application and applet only the rows and columns it needs.

You'll learn how to create and access a view from within your application in this chapter. You'll also learn how to create views with multiple columns, multiple rows, and multiple tables.

A View Is a Table without Chairs

You created a table that contains orders from each store in your retail store chain. However, you prefer that each store manager sees only those orders received by his or her store. In other words, no store manager is able to see another store's orders.

On the other hand, you want the central office staff to have access to all the orders in the table. And to further complicate matters, there is one table that contains all the orders. How can you limit a store manager's access to orders while providing full access to all the orders by the central office staff?

The answer is to create a view for each store manager. By creating a view, you can

- Limit access to rows that contain orders placed by a particular store
- Limit access to columns, such as the order number and amount
- Limit access to tables within the database

In addition to creating a view for each store manager, you can also create a view for members of the central office staff. For example, some staff members can have access to all rows and all columns except the column that contains the amount of the order. Other staff members who have access to your company's financial data could have a view that contains all rows and columns, including the amount of the order.

The advantages of using a view

You might be wondering if it's worth the time and effort necessary to create a view since all the data in a view is available from tables in the database. Yes, you'll find it worthwhile to create views for use with your Java database application.

Here are the benefits of creating and using a view:

- **Simplify tables:** You'll discover after building a few applications that you need only some of the data in the database. You can reduce the complexity of working with the database by creating a view of just the data your application needs.
- **Hide data:** A view acts like a security screen by making only selected columns and rows available to the application. You can exclude sensitive data from a view.

- **Enforce data integrity:** You can use the `CHECK OPTION` when creating a view to automatically enforce data integrity.
- **Make it easy to program:** Since a view already joins together tables into a virtual table, you don't have to include joins in your application's SQL expression. (See Chapter 17 for more about joins.)

The disadvantages of using a view

There is a downside to views that could pose a problem for your Java database application. You need to evaluate whether creating and using a view is advantageous to your application.

Here are the drawbacks to using a view:

- **Restrictions on inserting new rows and updating existing rows:** Since a view is a virtual table, it doesn't have any data itself. Instead, the data is in the tables that are used to create the view. This means there might be restrictions imposed by the underlying table that prevents the insertion of a row or update of a value in the underlying table (see "Modifying Views").
- **Performance degradation:** The SQL statement sent by your application to the DBMS can specify a view instead of a table. However, the DBMS translates the SQL statement into other SQL statements that are applied to the underlying tables of the view. The translation can decrease the response time for data to be returned to your application if the SQL statement contains a complex SQL expression and multiple underlying tables are used to create the view.

Try creating and using a view with your application. If you experience performance degradation, consider either reducing the number of tables used in the view (see "Using a View in Your Program") or abandoning the view and accessing the underlying tables directly by your application (see Chapter 13).

Using a View in Your Program

You should carefully plan the columns and tables that will become part of a view before you actually create the view. Although this should be obvious, I have a few colleagues who feel planning a view is a waste of time. Instead of planning, they plow into writing Java code to create the view.

The major disadvantage of not planning for the components of a view is that you might find that a view is missing required columns, rows, or tables after you begin to use it. This result doubles your work because you'll need to drop the view (see "Dropping a View: Whoops!") and then create a new view that contains the missing pieces.

Planning a view is a straightforward process. Here's what you need to do:

- **Review all the tables in the database:** The database administrator can help you do this if you are using a database management system (DBMS) that isn't on your hard drive.
- **List the columns that you need for the view:** I usually write down the names of the columns and their associated tables on a piece of paper.
- **List the rows that you need for the view:** You may want to restrict access to specific rows, such as rows that have a specific customer number or store number. Write down on a piece of paper the values (for example, customer number) used to isolate rows that can be accessed.

Once you've identified tables, columns, and rows for the view, you're ready to write your Java database application to create a view.

Setting up the tables

Before I show you how to create a view, you'll need to create a couple of tables and populate those tables with data. Begin by creating an Orders table, as shown in Figure BC1-1, and then create a Products table, shown in Figure BC1-2.

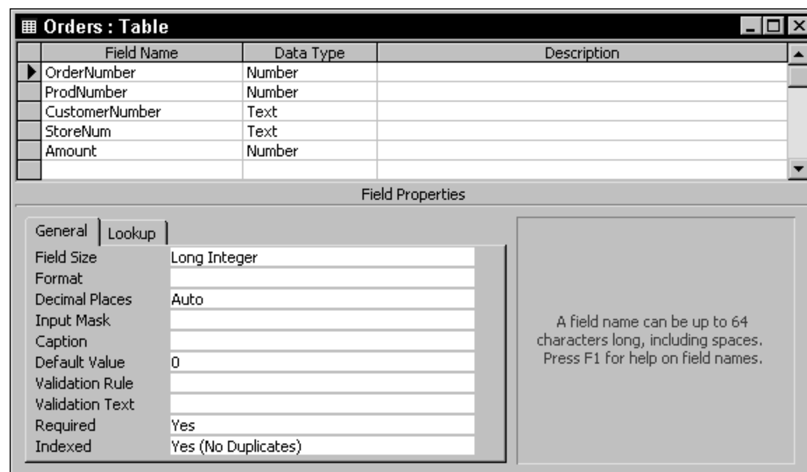


Figure BC1-1: Create an Orders table with the columns described in this figure.

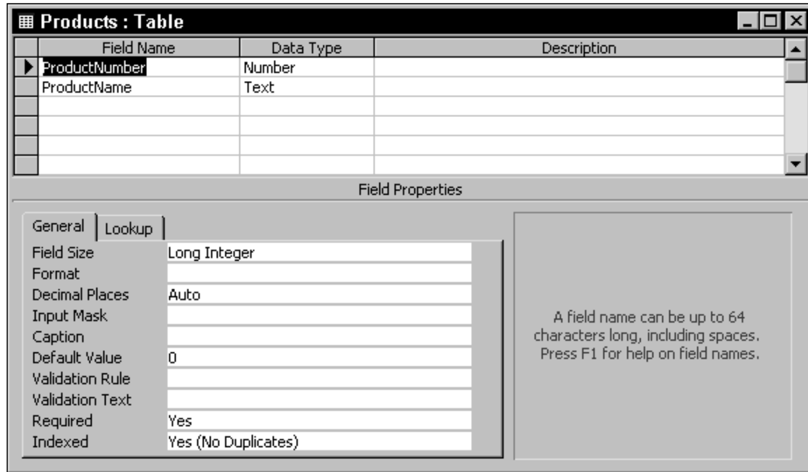
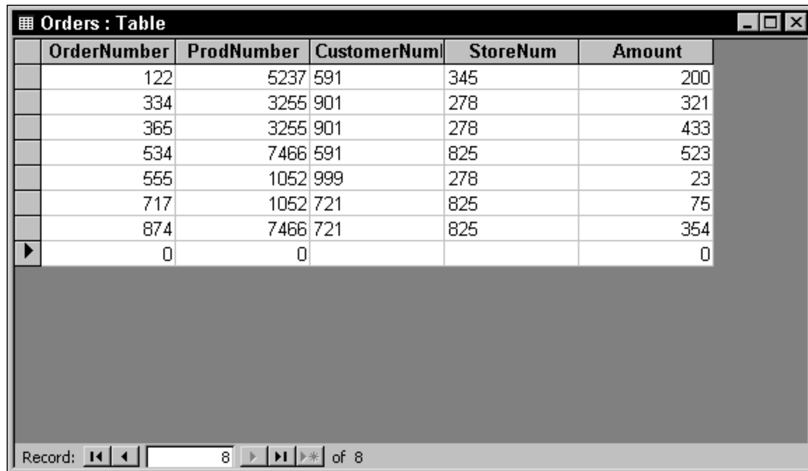


Figure BC1-2: Create a Products table with columns shown in this figure.

Once you create the tables, insert new rows into those tables. Figure BC1-3 contains the row I inserted into the Orders table, and Figure BC1-4 shows rows for the Products table. You can insert your own data into these tables; however, when you run the code examples in this chapter you'll have different results than the results I show you.



Orders : Table

OrderNumber	ProdNumber	CustomerNum	StoreNum	Amount
122	5237	591	345	200
334	3255	901	278	321
365	3255	901	278	433
534	7466	591	825	523
555	1052	999	278	23
717	1052	721	825	75
874	7466	721	825	354
0	0			0

Record: 8 of 8

Figure BC1-3: Insert rows into the Orders table.

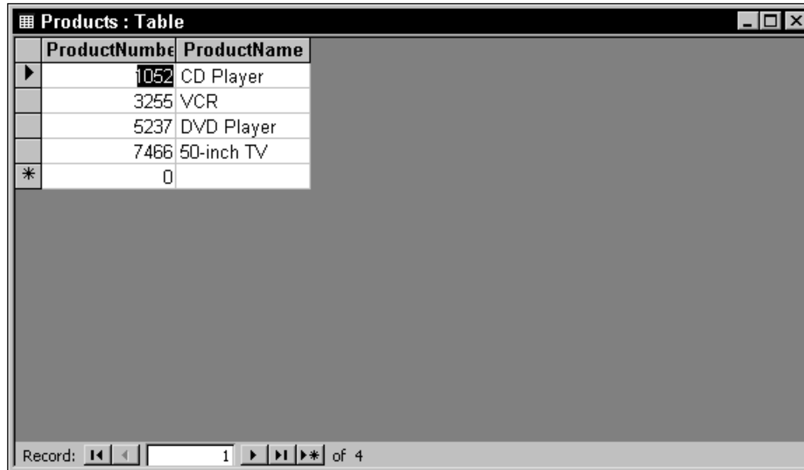


Figure BC1-4: Insert rows into the Products table.

Creating your first view: Selected rows

Use the `CREATE VIEW` statement that contains the name of the view and the definition of the view to create a view. The `AS` modifier separates the name of the view from the definition. The SQL expression in the following program illustrates this.

The definition of a view is an SQL expression that returns one or more columns of rows that meet the selection criteria. In Listing BC1-1, I define the Store278 view as all the columns of the Orders table and rows from the Orders table where the value of the StoreNum column equals 278. This means that the view looks like the Order table but contains only rows related to store number 278.

You'll see an SQL error message appear after you run this program. The error tells you no ResultSet returns. Ignore this message because the application wasn't expecting a ResultSet when creating a view.

Listing BC1-1: Creating a View

```
        import java.sql.*;

public class viewtst
{
   private Connection Database;

   public viewtst()
   {
      String url = "jdbc:odbc:customers";
      String userID = "jim";
```

```
          String password = "keogh";
          Statement DataRequest;
          ResultSet Results;

          try {
              Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

              Database =
            DriverManager.getConnection(url,userID,password);
          }
          catch (ClassNotFoundException error) {
              System.err.println("Unable to load the JDBC/ODBC
            bridge." + error);
              System.exit(1);
          }
          catch (SQLException error) {
              System.err.println("Cannot connect to the
            database." + error);
              System.exit(2);
          }

          try {
              String query = " CREATE VIEW Store278 AS " +
                             " SELECT  * " +
                             " FROM Orders  " +
                             " WHERE StoreNum  = '278'";
              DataRequest = Database.createStatement();
              Results = DataRequest.executeQuery (query);
              DataRequest.close();
          }
          catch ( SQLException error ){
              System.err.println("SQL error." + error);
              System.exit(3);
          }
          shutDown();
       }

    public void shutDown()
    {
       try {
           Database.close();
       }
       catch (SQLException error) {
           System.err.println( "Cannot break connection." +
            error ) ;
       }
    }
    public static void main ( String args [] )
    {
       final viewtst link = new viewtst();
               System.exit ( 0 ) ;
    }
}
```

Let's review the previous program beginning with the `main()` method at the bottom of the list. This is where the program begins. The `main()` method creates an object called `link` of the `viewtst` class, which is defined at the top of the code listing.

When the program creates the object, it also automatically executes the constructor of the `viewtst` class. The `viewtst()` constructor is the first method defined in the `viewtst` class definition. The constructor begins by creating `String` objects that are assigned the `url`, `userID`, and `password` that are necessary to access the database. The constructor also creates objects of the `Statement` class and the `ResultSet` class. The `Statement` object (`DataRequest`) sends the SQL statement to the DBMS. The `ResultSet` object references the ResultSet returned by the DBMS to the program.

Next, the constructor loads the JDBC/ODBC bridge (`Class.forName()`) and opens a connection with the database (`getConnection()`). If either attempt fails, one of two `catch{}` blocks trap the error. The first `catch{}` block traps errors that occur when loading the bridge. The other `catch{}` does the same for errors that happen when connecting to the database. In both cases, the `catch{}` block displays the error message on the screen and exits the program.

Once the bridge is loaded successfully and connection is made to the database, the constructor creates the SQL expression that creates the view. (See "Creating your first view: Selected rows" for an explanation of the SQL expression.)

The program creates an SQL statement (`createStatement()`) and uses it to execute the SQL expression (`executeQuery()`). The program returns a reference to the ResultSet, and the reference is assigned to the `Result` object. No ResultSet is expected because the SQL expression doesn't request data from the DBMS when creating a view. The SQL statement then closes (`close()`) once the SQL expression executes.

The `catch{}` block traps any errors that occur during the execution of the SQL expression, displays an error message onscreen, and exits the program.

Following execution of the SQL expression, the constructor calls the `shutDown()` method. The `shutDown()` method closes the database connection (`close()`). The `catch{}` block displays an error message if the connection can't close.

After the constructor has finished executing, the program returns to the `main()` method where the program terminates (`exit(0)`).

Verifying that you have a view

Nothing much happens when you create a view – at least nothing that you can see. Unlike an SQL expression that returns data from the DBMS, you don't see data when you create a view. Therefore, you need to write a simple program that tests whether the view was created successfully.

I used the program in Chapter 14 to verify that the DBMS created the view I told it to. You can use the same program to verify that your view was created. Here's what you need to do:

1. **Enter Listing 14-1 from Chapter 14.**
2. **Replace the `try{}` block that contains the SQL expression with the following code segment:**

```
        try {
            String query = "Select  StoreNum,
        OrderNumber " +
                           "FROM Store278 ";
            DataRequest = Database.createStatement();
            Results = DataRequest.executeQuery (query );
            DisplayResults (Results );
            DataRequest.close();
        }
```

   Notice that I use the name of the view (Store278) in the `FROM` clause in place of the name of a table. This means that the DBMS returns the value of the StoreNum column and the value of the OrderNumber column from the view Store278.

3. **Replace the `DownRow()` method with the following `DownRow()` method, which copies the store number and order number values from the ResultSet and displays them onscreen:**

```
    private  void DownRow ( ResultSet DisplayResults )
            throws SQLException
    {
       String Store278= new String();
       long OrderNumber;
       Store278 = DisplayResults.getString ( 1 ) ;
       OrderNumber = DisplayResults.getLong ( 2 ) ;
      System.out.println(Store278 + " " + OrderNumber);
    }
```

4. **Recompile and run the program. Figure BC1-5 shows the results that you'll see onscreen.**

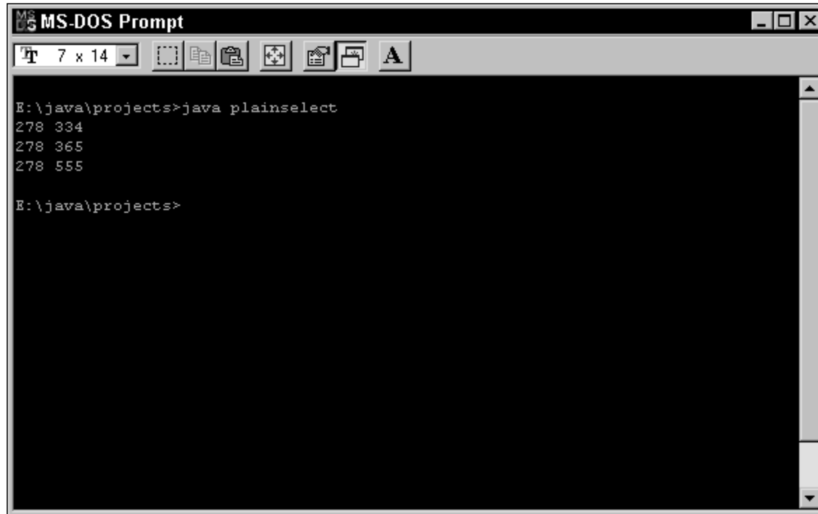


Figure BC1-5: Only store numbers and order numbers for Store 278 are displayed when using a view.

Showing the columns of your choice

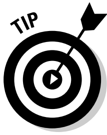
You can specify the columns used in the view by including column names in the definition of a view. People using your database will be able to access only those column names that appear in the view's definition.

Listing BC1-2 illustrates how to limit a view to specific columns. In this example, I create a view called StoreProd based on rows and columns in the Orders table. The StoreNum and ProdNumber rows from the Orders table define the view.

I haven't restricted the rows that can be viewed using the StoreProd view. So when an application uses the view, all the rows in the Orders table are available. However, the application can access only the StoreNum and the ProdNumber columns for those rows.

Listing BC1-2: Showing Columns of Your Choice

```
try {
    String query = " CREATE VIEW StoreProd AS " +
                   " SELECT  StoreNum, ProdNumber "
  +
                   " FROM Orders  ";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DataRequest.close();
}
```



The DBMS returns an SQL error if an application attempts to access columns or rows that are outside of the definition of a view.

You can use the previous code segment and other code segments that I use in this chapter with Listing BC1-1, the first program in the chapter. Simply substitute the `try{}` block in the program that contains the SQL expression with code segment. Recompile and run the program.

Picking and Choosing the Right View for You

A view restricts access to a specified set of columns and rows from one or more tables in the database. These restrictions are commonly referred to as a horizontal view and a vertical view because of the design of the view.

Here's how they work:

- **Horizontal view:** Provides access to all the columns in one or more tables, but limits access to a set of rows.
- **Vertical view:** Provides access to all rows in one or more table, but limits access to a set of columns.

Typically a join creates horizontal and vertical views from more than one table. A *join* is where the DBMS combines rows of two tables using a value that is common to a column in both tables (see Chapter 17).

Creating a horizontal view

Listing BC1-3 illustrates how to create a horizontal view by using two tables. Here I tell the DBMS to create a view called ProdDesc278. The view contains all the columns from the Orders table and from the Products table.

The `WHERE` clause contains two expressions. The first expression joins together rows of the Orders table and rows of the Products table using the value of the ProdNumber and ProductNumber columns. The other expression restricts the view to rows where the value of the StoreNum column is 278.

This means that a Java application that uses the ProdDesc278 view can access columns of both the Orders table and the Products table as if these columns appeared in the same table. However, the application is only able to access rows where the store number is 278.

Listing BC1-3: Creating a Horizontal View

```
try {
    String query = " CREATE VIEW ProdDesc278 AS " +
                   " SELECT  * " +
                   " FROM Orders, Products  " +
                   " WHERE ProdNumber =
  ProductNumber" +
                   "     AND StoreNum = '278'";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DataRequest.close();
}
```

Creating a vertical view

You create a vertical view by joining together relative tables and then specifying the column names from these tables that the view can access.

In Listing BC1-4, I create the view by joining together the Orders table and the Products table by using the value in the ProdNumber and ProductNumber columns. The view contains three columns: StoreNum, ProdNumber, and ProductName.

An application that uses the view can access all the rows contained in the Orders table and Products table. However, the application is limited to accessing the store number, product number, and product name data from those rows.

Listing BC1-4: Creating a Vertical View

```
try {
    String query = " CREATE VIEW ProdDesc AS " +
                   " SELECT  StoreNum, ProdNumber,
  ProductName " +
                   " FROM Orders, Products  " +
                   " WHERE ProdNumber =
  ProductNumber";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DataRequest.close();
}
```

Getting Organized: Grouping Views

An objective of using a view is to simplify data access by a Java database application. Instead of building complex SQL expressions in an application,

these SQL expressions can be made part of the definition of a view. So, the programmer who uses the view doesn't have to be concerned with formulating the complex SQL expression because it's already applied when the programmer uses the view.

Let's say that an application needs to retrieve data from multiple tables and organize the data into groups, such as by store number or product number. The definition of a view can join tables and group data. This leaves the programmer only to use the view and specify the rows that the application needs.

When a definition of a view groups data, the view is called an *ordered view.* You can use an ordered view to

- Summarize data
- Group data (see Chapter 18)
- Order data (see Chapter 18)

It's preferred to place complex SQL expressions in the definition of a view rather than place the SQL expression in an application that uses the view — as long as the complexity doesn't impact performance of the application. A complex view definition can consume processing time; therefore, you must balance the need to simplify complexity for the application and the need to efficiently process the SQL expression without using a view.

You create an ordered view by including the `ORDER BY` clause in the definition of the view. I illustrate this technique in Listing BC1-6. I define the GroupProdDesc view as having all the rows in the Orders table and the Products table and having the store number, product number, and product name columns.

The product number found in both tables joins the tables and organizes the data.

Listing BC1-6: Creating an Ordered View

```
try {
    String query = " CREATE VIEW GroupProdDesc AS "
  +
                   " SELECT  StoreNum, ProdNumber,
  ProductName " +
                   " FROM Orders, Products  " +
                   " WHERE ProdNumber =
  ProductNumber" +
                   " ORDER BY ProdNumber  ";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DataRequest.close();
}
```

Modifying Views

You can instruct the DBMS to modify the underlying tables that are associated with a view by modifying the view itself. And you can do this without having to reference tables in the database. You can modify a view using the same statements that are used to modify a table – only you use the name of the view in place of the table name.

The DBMS translates all modifications to a view into instructions to modify the underlying table(s) accordingly. The three types of modifications that can be made to a view are

- **Update:** Update substitutes a value in a column or columns with a new value specified in the `UPDATE` statement (see "Updating a view").
- **Insert:** Insert places a new row in an underlying table or tables of a view (see "Inserting a row into a view")
- **Delete:** Delete removes a specified row from an underlying table or tables (see "Deleting a row from a view").

Following the rules for updating a view

The DBMS is able to translate modifications made to a view into SQL statements to modify the underlying tables only if each row of the view corresponds to a row in the tables. Modifications don't execute if there isn't a one-to-one relationship between the rows of the view and the tables.

Here are rules that determine whether the DBMS will modify underlying tables of a view:

- Only column names can be used in the select list. Calculations, expressions, and column functions are prohibited.
- The SQL expression must not use the `DISTINCT` modifier because duplicate rows cannot be excluded when a view is modified.
- You must exclude `GROUP BY` and `HAVING` clauses from the expression.
- You must exclude subqueries from the `WHERE` clause (see Chapter 20).
- You must specify a table in the `FROM` clause of the view definition that the application has a right to modify.

Updating a view

You can update a view by using the same technique as is used to update values in a table (see Chapter 15) – that is, by using the UPDATE statement. Listing BC1-7 illustrates the UPDATE statement. Here I tell the DBMS to change the current value of the Amount column to 700.

The UPDATE statement uses the name of the view, which is Store278. The SET modifier specifies the name of the column and the value that is to be placed into that column. And the WHERE clause contains the expression that identifies the row that is to be updated.

Listing BC1-7: Updating a View

```
try {
    String query = " UPDATE Store278 " +
                   " SET Amount = 700 " +
                   " WHERE OrderNumber  = 334 ";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DataRequest.close();
}
```

Inserting a row into a view

The INSERT INTO statement inserts a new row into the underlying tables of a view (see Chapter 12). Listing BC1-8 places a new row into the underlying tables of the Store278 view.

The parentheses following the name of the view contain the list of columns contained in the view. The VALUES modifier is followed by values that the DBMS will place into the new row. The order of the columns and the order of the values must be synchronized. That is, value 325 is placed into the OrderNumber column, 9545 into the ProdNumber column, and so on.

Listing BC1-8: Inserting a Row into a View

```
try {
    String query = " INSERT INTO Store278 " +
       " (OrderNumber, ProdNumber, CustomerNumber,
     StoreNum, Amount) " +
       " VALUES (325, 9545 ,301 ,278 ,400) ";
       DataRequest = Database.createStatement();
       Results = DataRequest.executeQuery (query);
       DataRequest.close();
    }
```

Deleting a row from a view

You can delete a row from the underlying table of a view by using the DELETE FROM statement (see Chapter 16). The DELETE FROM statement specifies the name of the view, which is Store278 in the following code segment.

The SQL expression also requires a WHERE clause that contains an expression used to identify the row that is to be deleted. In Listing BC1-9, the row that contains order number 325 will be removed from the relative table by the DBMS when this code segment executes.

Listing BC1-9: Deleting a Row from a View

```
try {
    String query = " DELETE FROM  Store278 " +
                   " WHERE OrderNumber = 325  " ;
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DataRequest.close();
}
```

Dropping a View: Whoops!

You can use the DROP statement to remove any view that your Java database application creates. The DROP statement is similar to the DROP statement used to delete a table from the database (see Chapter 16).

The DROP statement has two modifiers that tell the DBMS how to handle dependent views. A *dependent view* is a view that has another view as one of its underlying components rather than a table. Therefore, you must understand the effect dropping a view has on other views before you drop the view.

The two DROP modifiers are

- **CASCADE:** The CASCADE modifier tells the DBMS to drop the specified view and any view that is dependent on the specified view.
- **RESTRICT:** The RESTRICT modifier tells the DBMS to drop only the specified view if no other views are dependent on the specified view.

Always consult with your database administrator before dropping a view so you understand the impact, if any, of dropping a view. The database administrator can identify other views and applications that also use the view.

You use the DROP VIEW statement to drop a view from the database. The name of the view must follow the DROP VIEW statement, as is shown in Listing BC1-10. You have the option of using a DROP modifier with the DROP VIEW statement.

In this example, I use the CASCADE modifier to tell the DBMS to drop the view and dependant views. The DBMS drops the view without considering dependent views if the DROP VIEW statement doesn't use a modifier.

You use the RESTRICT modifier in the same position as the CASCADE modifier in the SQL expression.

Some DBMSs, such as Microsoft Access, may not recognize the DROP VIEW command. Consult with your database administrator to determine a comparable command for the DBMS used with your application.

Listing BC1-10: Dropping a View

```
try {
    String query = " DROP VIEW  Store278 CASCADE";
    DataRequest = Database.createStatement();
    Results = DataRequest.executeQuery (query);
    DataRequest.close();
}
```

Bonus Chapter 2

I've Got a Deal for You: Transaction Processing

In This Chapter

- Creating a transaction
- Committing a transaction
- Rolling back a transaction
- Dealing with deadlocks

Although most of us associate the term *transaction* with making a purchase, a transaction is any set of tasks that are performed in a particular order to achieve a result. For example, an SQL statement embedded into your Java database application is a transaction because the statement defines a series of tasks that the database management system (DBMS) performs.

A transaction can succeed or fail, it but cannot almost succeed or almost fail. It's either all or nothing. Either you make the purchase or you don't make the purchase. Either the SQL statement executes or it doesn't execute.

In reality, any transaction can fail during the transaction process. Whenever a task fails in a transaction, you must reverse the steps of the transactions that have been successfully completed.

This chapter shows you how to use the `COMMIT` and `ROLLBACK` statements within your Java database application to give your application better control over critical transactions.

Either COMMIT or ROLLBACK, Son

You and I transact business every day, yet we probably don't give much thought to the transaction process because transactions have become commonplace. However, transactions follow a prescribed set of steps, each of which must be completed successfully before the transaction itself is completed.

Each step in a transaction has an opportunity to fail, which means the previous steps must be undone. If you go shopping at the supermarket, for example, here would be the transaction steps you'd probably follow:

1. You place an item from the shelf into your shopping cart.
2. You place items from your shopping cart onto the checkout counter.
3. The clerk scans an item into the cash register.
4. The clerk repeats Step 3 until all the items in your shopping cart are scanned.
5. The clerk tallies the prices of all the goods and presents you with the total.
6. You give the clerk payment for the items, and the clerk gives you change if necessary.

At any point in the transaction, you or the clerk can decide to cancel the transaction. However, once the transaction is interrupted, you must reverse the steps prior to the interruption. For example, the clerk must subtract all the items scanned into the cash register if you are unable to pay for the merchandise.

An SQL transaction

A transaction within your application consists of one or more related SQL statements that are executed sequentially. Let's say that you've written a Java database application that enables a sales representative to enter a customer's order. Your application places data associated with a customer's order into two tables in the database. For example, the customer's name and address go into the customers table, and data about the order goes into the Orders table.

Therefore, your application must execute two SQL statements, which are related to each other. One statement inserts a row into the customers table, and the other statement inserts a row into the Orders table. Both statements are related to each other because they pertain to the same customer.

Terms of commitment

A transaction is a block of SQL statements or at least one SQL statement that is terminated with the `COMMIT` statement. The `COMMIT` statement tells the DBMS that the transaction is completed and SQL statements that comprise the transaction have executed.

Using the `ROLLBACK` statement can reverse any SQL statement that executes prior to the `COMMIT` statement. The `ROLLBACK` statement basically tells the DBMS that your application made an error that needs to be reversed.

When the DBMS receives the `ROLLBACK` statement, it undoes the previously executed SQL statements. This restores the database to the original state prior to running the SQL statements.

Here is an example to illustrate the two statements. Let's say that your application prompts a sales rep to enter a new customer order. The sales representative proceeds to enter the customer's name and address, which your application immediately enters into the customers table.

Next, your application prompts the sales rep to enter order information. However, the sales rep is unclear on an item the customer ordered, so he cancels the ordering process.

Your application needs to back out the customer data the sales rep entered into the customers table. You would do this by issuing the `ROLLBACK` statement to the DBMS. This causes the DBMS to remove the customer's data from the database.

On the other hand, your application would issue the `COMMIT` statement if the sales representative finishes entering the customer's name, address, and order data. At this point, the transaction completes.

Setting Up the Tables

Before trying the examples that I show you in this chapter, you need to create two tables: the customers table and the Orders table.

Figure BC2-1 contains the table definition for the customers table, and Figure BC2-2 contains the table definition for the Orders table. Once you've created your tables, enter data into the tables. Figure BC2-3 shows you the rows that I entered into the customers table, and Figure BC2-4 shows the data I used for the Orders table.

customers : Table

Field Name	Data Type	Description
CustomerNumber	Number	Customer Number
FirstName	Text	Customer First Name
LastName	Text	Customer Last Name
Street	Text	Customer Street
City	Text	Customer City
ZipCode	Text	Customer Zip Code

Field Properties

General | Lookup

Field Size	Long Integer
Format	
Decimal Places	Auto
Input Mask	
Caption	
Default Value	0
Validation Rule	
Validation Text	
Required	Yes
Indexed	Yes (No Duplicates)

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

Figure BC2-1: Create a table called customers.

Orders : Table

Field Name	Data Type	Description
OrderNumber	Number	
ProdNumber	Number	
CustomerNumber	Text	
StoreNum	Text	
Amount	Number	

Field Properties

General | Lookup

Field Size	Long Integer
Format	
Decimal Places	Auto
Input Mask	
Caption	
Default Value	0
Validation Rule	
Validation Text	
Required	Yes
Indexed	Yes (No Duplicates)

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

Figure BC2-2: Create a table called Orders.

customers : Table

CustomerNur	FirstName	LastName	Street	City	ZipCode
591	Anne	Smith	65 cutter Street	Woodridge	04735
721	Bart	Adams	15 W. Spruce	River Ville	05213
845	Tom	Jones	35 Pine Street	West Town	07660
0					

Record: 4 of 4

Figure BC2-3: Insert rows into the customers table.

Orders : Table

OrderNumber	ProdNumber	CustomerNuml	StoreNum	Amount
122	5237	591	345	200
555	1052	999	278	23
717	1052	721	825	75
874	7466	721	825	354
0	0			0

Record: 5 of 5

Figure BC2-4: Insert rows into the Orders table.

Using COMMIT in Your Program

The `COMMIT` statement is included in the last portion of the SQL expression in your Java database application. Listing BC2-1 tells the DBMS to insert one row into the customers table and one row into the Orders table (see Chapter 12).

Once the DBMS inserts the rows into both tables, the program issues the `COMMIT` statement, which tells the DBMS that the transaction is completed.

Listing BC2-1: Using COMMIT

```
import java.sql.*;

   public class inserttst
   {
      private Connection Database;

      public inserttst()
      {
         String url = "jdbc:odbc:customers";
         String userID = "jim";
         String password = "keogh";

         try {
             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");

             Database =
           DriverManager.getConnection(url,userID,password);
         }
         catch (ClassNotFoundException error) {
             System.err.println("Unable to load the JDBC/ODBC
           bridge." + error);
             System.exit(1);
         }
         catch (SQLException error) {
             System.err.println("Cannot connect to the
           database.");
             System.exit(2);
         }

         Statement DataRequest;
         try {
             String query = "INSERT INTO customers  " +
                            " VALUES (901,'Mary','Smith', ' 5
           Maple Street', 'SunnySide','08513') " +
                            " INSERT INTO Orders   " +
                            " VALUES (334, 3255, '901',
           '278', 700) ";


   String query2 = " COMMIT ";
             DataRequest = Database.createStatement();
             DataRequest.executeQuery (query);
             DataRequest.close();

             DataRequest = Database.createStatement();
             DataRequest.executeQuery (query2);
             DataRequest.close();
         }

catch ( SQLException error ){
             System.err.println("SQL error." + error);
```

```
         }
      }

   public static void main ( String args [] )
   {
      final inserttst link = new inserttst();
            System.exit ( 0 ) ;
   }
}
```

The main() method, located near the end of the listing, is where the program begins. It's there that the program creates an object of the inserttst class. The inserttst class itself is defined at the top of the listing.

Once the program creates the object of the inserttst class, it runs the inserttst class constructor (inserttst()). The constructor creates three String objects that are assigned the url of the database, the userID, and the password necessary to connect to the database. The constructor also creates a Statement object to execute the SQL statement and a ResultSet object to reference the ResultSet returned by the DBMS.

Next, the constructor loads the JDBC/ODBC bridge (Class.forName()) and opens the database connection (getConnection()). One of two catch{} blocks traps any errors that occur during these tasks. The first catch{} block displays an error message and exits the program if the JDBC/ODBC bridge is unable to load. The second catch{} block also displays an error message and exits the program if the connection to the database cannot be opened.

The constructor then creates two SQL expressions. The first contains two INSERT statements and the second contains the COMMIT statement. The constructor also creates an SQL statement (createStatement()). The SQL statement is associated with the SQL expression and is sent to the DBMS for execution (executeQuery()). Afterwards, the statement closes (close()).

Again, the catch{}block traps any errors that occur when the SQL statement executes and displays an error message onscreen.

After the constructor is finished executing, the program returns to the main() method where the program terminates (exit()).

Using ROLLBACK in Your Program

Your Java database application can reverse modifications made to the database by sending an SQL statement containing the ROLLBACK statement. The ROLLBACK statement returns the database to the state prior to executing the original SQL statement.

Not all DBMSs work alike

Practically all DBMSs have transactional-processing capabilities. However, you'll find that some DBMSs use variations on the COMMIT and ROLLBACK statements. For example, here's how Sybase handles transactional processing:

- Transactions start with the BEGIN TRANSACTION statement.
- Transactions are committed using the COMMIT TRANSACTION statement.
- Transactions are rolled back using the ROLLBACK TRANSACTION statement.

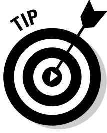
Therefore, you should consult with your database administrator to determine the proper COMMIT and ROLLBACK statements to use with your Java database application.

Listing BC2-2 illustrates how to use the ROLLBACK statement in your program. In this example, the ROLLBACK statement is placed in the SQL expression of the catch{} block. The catch{} block traps SQL errors that occur in the try{} block.

If the DBMS successfully inserts data into the customers table but is unable to insert data into the Orders table, it sends the application an SQL error that is detected by the catch{} block. The catch{} block then tells the DBMS to rollback, or undo, the new row that was placed into the customers table.

Listing BC2-2: Using ROLLBACK

```
try {
    String query = "INSERT INTO customers   " +
                   " VALUES (901,'Mary','Smith', ' 5
  Maple Street', 'SunnySide','08513') " +
                   " INSERT INTO Orders    " +
                   " VALUES (334, 3255, '901',
  '278', 700) ";
    DataRequest = Database.createStatement();
    DataRequest.executeQuery (query);
    DataRequest.close();
}
catch ( SQLException error ){
    System.err.println("SQL error." + error);
    try{
       String query = "ROLLBACK ";
    DataRequest = Database.createStatement();
    DataRequest.executeQuery (query);
    DataRequest.close();
    }
}
```



Real-world Java database applications modify more than one row at a time, which increases the likelihood that an SQL error could occur. If an error occurs, your application must roll back rows that have been modified during the transaction.

Avoiding the Dreaded Deadlocks

Any number of errors can occur while your application is processing a transaction. One of the most common of these is being locked out of a table. A *lockout* occurs when another application accesses the same row or table your application needs to update.

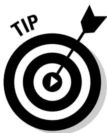
Some DBMSs institute a table lock whenever an application is modifying a row in the table, so that no other application can access the table while the row is being modified. Other DBMSs use a row lockout instead of a table lockout. A row lockout prevents access to the row that is being updated rather than the complete table. Check with your database administrator to determine which lockout scheme your DBMS uses.

When the DBMS encounters a lock during a transaction, the DBMS returns an SQL error to your application, indicating that access was denied. Your application makes another attempt to modify the row by resending the SQL statement or issues the `ROLLBACK` statement to start the transaction over again.

However, there can be occasions when your application and another application attempt to modify the same row of the same table at the same time. This is called a deadlock because neither application gains access to the row. Instead, both applications are denied access.

A deadlock can seem never ending. Typically, both applications resend their SQL statements after being denied access. The problem arises because these events occur simultaneously for both applications.

Some DBMSs reduce the likelihood of recurring deadlocks by slightly delaying responses to applications that are deadlocked. That is, notice of a deadlock is sent to one application and a second later a similar notice is sent to the other deadlocked application. This effectively staggers the sequence of resending the SQL statement.



Contact your database administrator whenever your application encounters an endless deadlock. He or she can identify applications that are deadlocked and take actions to break the deadlock.