COP 4610L: Applications in the Enterprise Spring 2005

Introduction to JDBC – Part 1

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CSB 242, 823-2790
http://www.cs.ucf.edu/courses/cop4610L/spr2005

School of Electrical Engineering and Computer Science
University of Central Florida

Introduction to JDBC

- JDBC was originally an acronym for Java Data Base Connectivity. Sun marketing now states this is no longer an acronym but the official name.
- JDBC is made up of about two dozen Java classes in the package `java.sql`. These classes provide access to relational data stored in a database or other table-oriented forms (like Excel, etc.).
- JDBC allows the programmer to use modern database features such as simultaneous connections to several databases, transaction management, precompiled statements with bind variables, calls to stored procedures, and access to metadata in the database dictionary.
- JDBC supports both static and dynamic SQL statements.
- The evolution of JDBC is shown on the next slide.

Evolution of JDBC

JDBC Version	Bundled with	Package Name	Contents
JDBC 1.0 (previously called 1.2)	JDK 1.1	java.sql	Basic Java client to database connectivity.
JDBC 2.0 core API	JDK 1.2 and later	java.sql	Added features such as scrollable result sets, batch updates, new data types for SQL-3, and programmable updates using the result set.
JDBC 2.0 optional API	J2EE and later	javax.sql	Can be downloaded from www.java.sun.com/products/jdbc/ . Contains database server-side functionality. Prepares the ground for the use of database-aware Java beans.
JDBC 2.1 optional API	Not bundled	javax.sql	Incremental improvement and additions over the 2.0 API.
JDBC 3.0 core API	JDK 1.4 and later	java.sql	Adds support for connection pooling, statement pooling, and a migration path to the Connector Architecture.

Connecting To A Database

- A database works in the classic client/server fashion. There is one database and many clients talk to it. (Larger applications may have multiple database, but they can be considered independently for our purposes.)
- As we've seen in the earlier sections of notes dealing with networking, the clients are typically remote systems communicating over TCP/IP networks.
- In a 2-tier system, the clients talk directly to the database while in a 3-tier system, the clients talk to a business logic server which in turn talks to the database. The business logic server would also contain server-side JDBC functionality.

Connecting To A Database (cont.)

- A JDBC driver is typically available from the database vendor of the database to which you wish to connect.
- There are several different kinds of drivers depending on whether it was written in Java or native code, or whether it talks directly to the database or through another database access protocol (such as Microsoft's ODBC). From an application programmer's point of view, none of this matters very much as long as you have a working JDBC driver, you really don't care how it works (although your client may if its too slow!).
- JDBC supports four categories of drivers which are detailed in the table on the next page.

JDBC Driver Types

Type	Description
1	**JDBC-to-ODBC Bridge Driver** – connects Java to a Microsoft ODBC (Open Database Connectivity) data source. This driver requires the ODBC driver to be installed on the client computer and configuration of the ODBC data source. This driver is used to allow Java programmers to build data-driver Java applications before the database vendor supplies a Type 3 or Type 4 driver. In general, this will not be used too much these days.
2	**Native-API, Part Java Drivers** – enable JDBC programs to use database-specific APIs (normally written in C or C++) that allow client programs to access databases via the Java Native Interface. This driver translates JDBC into database-specific code. Reasons for use are similar to Type 1.
3	**JDBC-Net Pure Java Drivers** – take JDBC requests and translate them into a network protocol that is not database specific. These requests are sent to a server, which translates the database requests into a database-specific protocol.
4	**Native-protocol Pure Java Drivers** – convert JDBC requests to database-specific network protocols, so that Java programs can connect directly to a database.

Some Popular JDBC Drivers

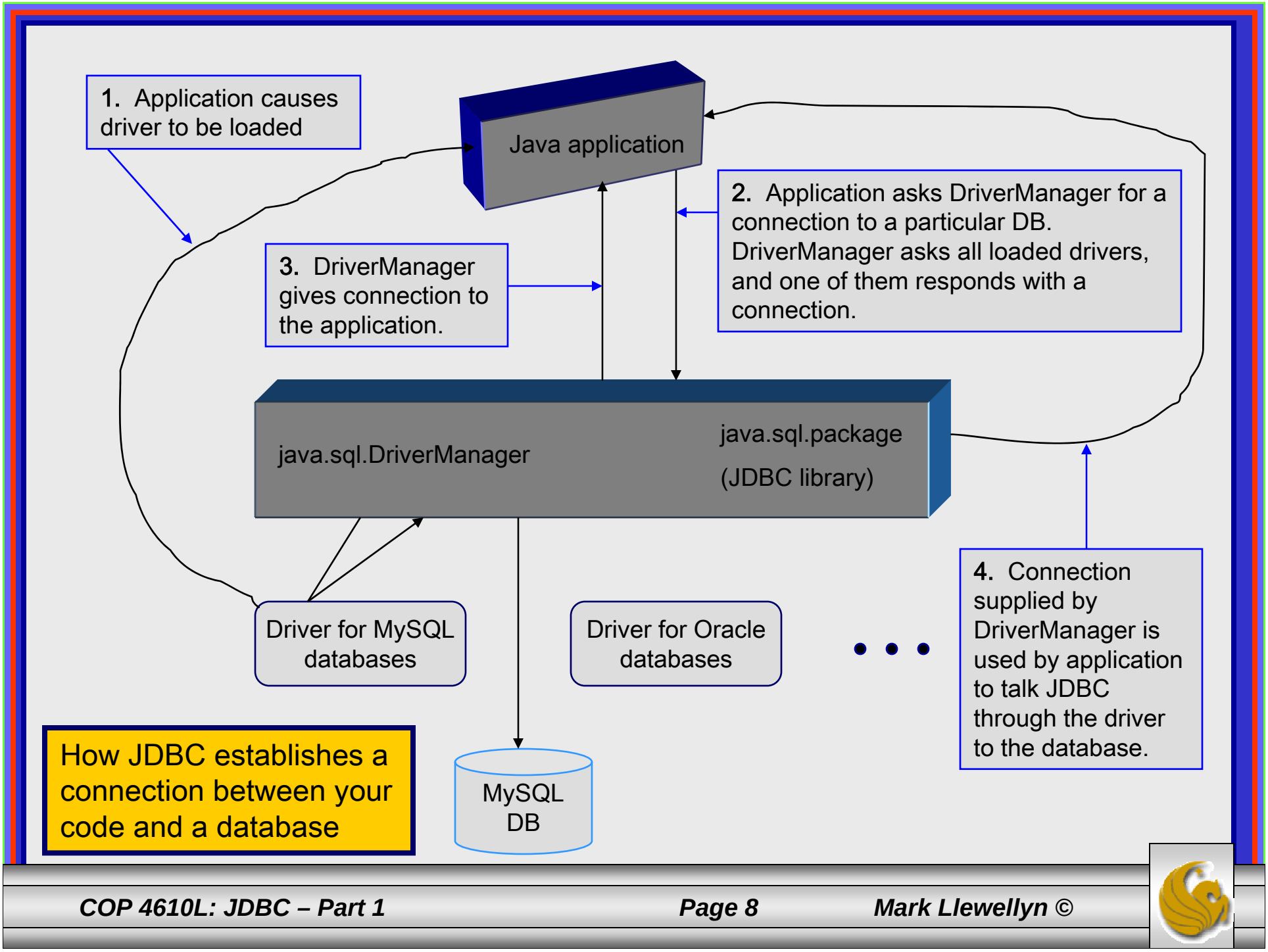
RDBMS	JDBC Driver Name

MySQL	Driver Name
	`com.mysql.jdbc.Driver`
	Database URL format:
	`jdbc:mysql//hostname/databaseName`

Oracle	Driver Name:
	`oracle.jdbc.driver.OracleDriver`
	Database URL format:
	`jdbc:oracle:thin@hostname:portnumber:databaseName`

DB2	Driver Name:
	`COM.ibm.db2.jdbc.net.DB2Driver`
	Database URL format:
	`jdbc:db2:hostname:portnumber/databaseName`

Access	Driver Name:
	`com.jdbc.odbc.JdbcOdbcDriver`
	Database URL format:
	`jdbc:odbc:databaseName`


1. Application causes driver to be loaded
Java application
2. Application asks DriverManager for a connection to a particular DB. DriverManager asks all loaded drivers, and one of them responds with a connection.
3. DriverManager gives connection to the application.
java.sql.DriverManager
java.sql.package
(JDBC library)
Driver for MySQL databases
Driver for Oracle databases
4. Connection supplied by DriverManager is used by application to talk JDBC through the driver to the database.
MySQL DB
How JDBC establishes a connection between your code and a database

Loading A JDBC Driver

- The first step (as illustrated in the previous slide) is to load a JDBC driver.
- If your application connects to several different types of databases, all of their respective drivers must be loaded.
- The Java statement to load a JDBC driver is:

```
Class.forName(" JDBC Driver Class ");
```

- You don't need to create an instance of the driver class. Simply getting the class loaded is enough. Each JDBC driver has a static initializer that is run when the class is loaded, and in that code the driver registers itself with the JDBC. The JDBC driver does about 90% of the work that is done in JDBC.

Establishing a Connection

- The second step involves the Java application requesting a connection to a database, using a string that looks like a URL as an argument.
- The JDBC library contains the class `java.sql.Connection` that knows how to use this string to guide it in its search for the correct database. As was shown in the table on page 7, the exact format of the pseudo-URL string will vary with each database, but it typically starts with "`jdbc:`" to indicate the protocol that will be used (just as "http:" indicates to a web server that you are using the hypertext transport protocol).
- The Java statement to connect to a database invokes the static method `getConnection(databaseURL)` in the `DriverManager` class:

```
Connection connection =
        DriverManager.getConnection(url, username, password);
```

Optional parameters

Establishing a Connection (cont.)

- Behind the scenes, the `DriverManager` calls every JDBC driver that has been registered, and asks it if the URL is one that it can use to guide it to its database.
- If the URL is properly presented, it will be recognized by at least one of the drivers.
- The first driver to connect to its database with this URL, username, and password, will be used as the channel of communication.
- The application program gets back a `Connection` object (strictly speaking it gets an object that implements the `Connection` interface).
- The session has now been established and the connection is now used for communication between the application program and the database.
- You can think of the `Connection` object as cable linking your application program to the database.

Establishing a Connection (cont.)

- Connecting to a database is a time consuming operation. As a result, most database have a way to share connections among several different processes. This arrangement is known as connection pooling.
- In summary:
  - Your application program knows which database it wants to talk to, and hence which database driver it needs to load.
  - The JDBC driver manager knows how to establish the JDBC end of a database connection.
  - The driver knows how to establish the database end of things.
  - The driver manager gives the application a connection into which you can pour standard SQL queries and get results.

Creating Statements

- If the `Connection` object can be viewed as a cable between your application program and the database, an object of Statement can be viewed as a cart that delivers SQL statements for execution by the database and brings the result back to the application program.
- Once a `Connection` object is created, you can create statements for executing SQL statements as follows:

```
Statement statement = connection.createStatement();
```

- At this point, you're now ready to begin issuing SQL commands to the database and getting back results. The table on the following page illustrates some of the methods contained in `java.sql.Connection`.

Selected Methods In `java.sql.Connection`

Method	Purpose
Statement createStatement()	Returns a statement object that is used to send SQL to the database.
PreparedStatement preparedStatement(String sql)	Returns an object that can be used for sending parameterized SQL statements.
CallableStatement prepareCall(String sql)	Returns an object that can be used for calling stored procedures.
DatabaseMetaData getMetaData()	Gets an object that supplied database configuration information.
boolean isClosed()	Reports whether the database is currently open or not.
void commit()	Makes all changes permanent since previous commit.rollback.
void rollback()	Undoes and discards all changes done since the previous commit/rollback.

Selected Methods In `java.sql.Connection`
(cont.)

Method	Purpose
void setAutoCommit (boolean yn)	Restores/removes auto-commit mode, which does an automatic commit after each statement. The default case is AutoCommit is on.
void close()	Closes the connection and releases the JDBC resources for the connection.
boolean isReadOnly()	Retrieves whether this Connection object is in read-only mode.
void setReadOnly(boolean yn)	Puts this connection in read-only mode as a hint to the driver to enable database optimizations.

Creating Statements (cont.)

- The methods illustrated in the previous table are invoked on the Connection object returned by the JDBC driver manager.
- The connection is used to create a Statement object.
- The Statement object contains the methods that allow you to send the SQL statements to the database.
- Statement objects are very simple objects which allow you to send SQL statements as Strings.
- Here is how you would send a select query to the database:

```
Statement myStmt = connection.createStatement();
ResultSet myResult;
myResult = myStmt.executeQuery( “SELECT * FROM bikes;”);
```

More on ResultSet later

Creating Statements (cont.)

- The different SQL statements have different return values. Some of them have no return value, some of them return the number of rows affected by the statement, and others return all the data extracted by the query.
- To handle these varied return results, you'll need to invoke a different method depending on what type of SQL statement you are executing.
- The most interesting of these is the SELECT statement that returns an entire result set of data.
- The following table highlights some of the methods in `java.sql.statement` to execute SQL statements.

Some Methods in java.sql.statement to Execute SQL Statements

SQL statement	JDBC statement to use	Return Type	Comment
SELECT	executeQuery(String sql)	ResultSet	The return value will hold the data extracted from the database.
INSERT, UPDATE, DELETE, CREATE, DROP	executeUpdate(String sql)	int	The return value will give the count of the number of rows changed , or zero otherwise.
Stored procedure with multiple results	execute(String sql)	boolean	The return value is true if the first result is a ResultSet, false otherwise.

Putting It All Together – A Simple Example

- Let's put all these pieces together and develop a Java application that will connect to our bikedb database, execute a query, and return the results.
- This application will show, in the simplest terms, how to load the JDBC driver, establish a connection, create a statement, have the statement executed, and return the results to the application.
- The code is shown on the next page with results on the following page.

```
// Very basic JDBC example showing loading of JDBC driver, establishing
// a connection, creating a statement, executing a simple SQL query, and
// displaying the results.
import java.sql.*;
public class SimpleJdbc {
  public static void main(String[] args)
    throws SQLException, ClassNotFoundException {
    // Load the JDBC driver
       Class.forName("com.mysql.jdbc.Driver");
       System.out.println("Driver loaded");
    // Establish a connection
    Connection connection = DriverManager.getConnection
           ("jdbc:mysql://localhost/bikedb", "root", "root");
    System.out.println("Database connected");
    // Create a statement
    Statement statement = connection.createStatement();
    // Execute a statement
    ResultSet resultSet = statement.executeQuery
      ("select bikename,cost,mileage from bikes");
    // Iterate through the result set and print the returned results
    while (resultSet.next())
      System.out.println(resultSet.getString(1) + "         \t" +
        resultSet.getString(2) + "       \t" + resultSet.getString(3));
    // Close the connection
    connection.close();
  }
}
```

Load JDBC driver

Establish connection specifying database, username, and password.

Create statement

Execute query

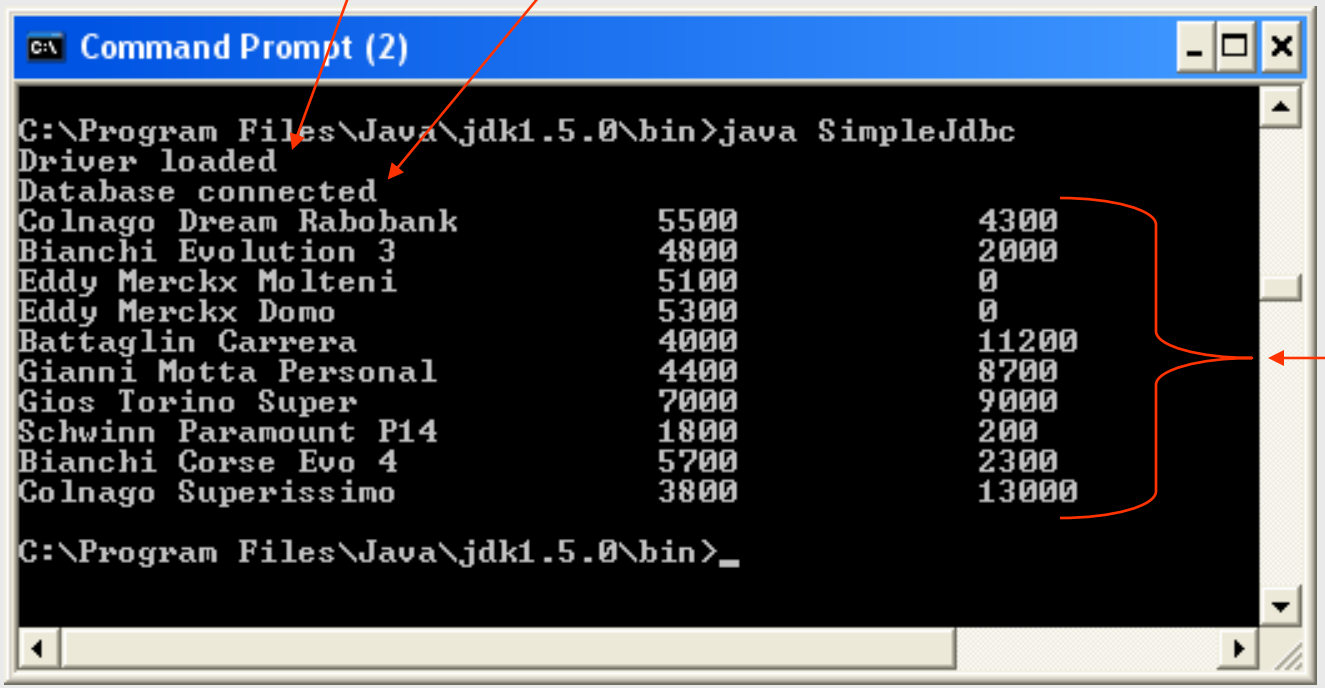
Iterate through resultSet

Close connection

SimpleJdbc.java – Execution Results

Driver successfully loaded

Connection successfully established



Query results printed

Result Sets

- A ResultSet object is similar to a 2D array. Each call to `next()` moves to the next record in the result set. You must call `next()` before you can see the first result record, and it returns false when there are no more result records (this makes it quite convenient for controlling a while loop). (Also remember that the `Iterator next()` returns the next object and not a true/false value.)
- The class ResultSet has "getter" methods `getBlob()`, `getBigDecimal()`, `getDate()`, `getBytes()`, `getInt()`, `getLong()`, `getString()`, `getObject()`, and so on, for all the Java types that represent SQL types for a column name and column number argument. Look at the documentation for `java.sql.ResultSet` for a complete listing of the methods.

Result Sets (cont.)

- A default `ResultSet` object is not updatable and has a cursor that only moves forward.
- Many database drivers support scrollable and updatable `ResultSet` objects.
  - Scrollable result sets simply provide a cursor to move backwards and forwards through the records of the result set.
  - Updatable result sets provide the user with the ability to modify the contents of the result set and update the database by returning the updated result set. NOTE: Not all updates can be reflected back into the database. It depends on the complexity of the query and how the data in the result set was derived. In general, base relation attribute values can be modified through an updatable result set. We'll see an example of this later.

ResultSet Constants for Specifying Properties

`ResultSet` static type constant	Description
TYPE_FORWARD_ONLY	Specifies that the ResultSet cursor can move only in the forward direction, from the first row to the last row in the result set.
TYPE_SCROLL_INSENSITIVE	Specifies that the ResultSet cursor can scroll in either direction and that the changes made to the result set during ResultSet processing are not reflected in the ResultSet unless the database is queried again.
TYPE_SCROLL_SENSITIVE	Specifies that the ResultSet cursor can scroll in either direction and that changes made to the result set during ResultSet processing are reflected immediately in the ResultSet.
`ResultSet` static concurrency constant	Description
CONCUR_READ_ONLY	Specifies that the ResultSet cannot be updated (i.e. changes to it will not be reflected into the database).
CONCUR_UPDATABLE	Specifies that the ResultSet can be updated (i.e. changes to it will be reflected into the database using ResultSet update methods).

ResultSet Examples

- The following two examples clarify the various constants which can be applied to result sets (assume that `connection` is a valid `Connection`).

```
//creates a ResultSet which is scrollable, insensitive
//to changes by others and updatable.
Statement stmt = connection.createStatement(
                 (ResultSet.TYPE_SCROLL_INSENSITIVE,
                  ResultSet.CONCUR_UPDATABLE );

//creates a ResultSet which is scrollable, sensitive
//to changes by others and updatable.
Statement stmt = connection.createStatement(
                 (ResultSet.TYPE_SCROLL_SENSITIVE,
                  ResultSet.CONCUR_UPDATABLE);
```

Another Example

- In the previous example, notice that in the output, there was no information about what the columns represent. The output appears to be just data rather than information.
- A more sophisticated example, will access the database and use the metadata to provide more significance to the results.
- In the next example, we do just that by retrieving metadata from the database to help with the display of the result set.

Simple JDBC Application – page 1

```
// Displaying the contents of the bikes table.
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class DisplayBikes{
   // JDBC driver name and database URL
   static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
   static final String DATABASE_URL = "jdbc:mysql://localhost/bikedb2";

   // launch the application
   public static void main( String args[] )  {
      Connection connection = null; // manages connection
      Statement statement = null; // query statement

      // connect to database bikes and query database
      try
      {
         Class.forName( JDBC_DRIVER ); // load database driver class
```

Type 4 JDBC driver (pure Java driver) to connect to MySQL RDBMs

Specify name of database to connect to as well as JDBC driver protocol.

See Note on Page 31

Simple JDBC Application – page 2

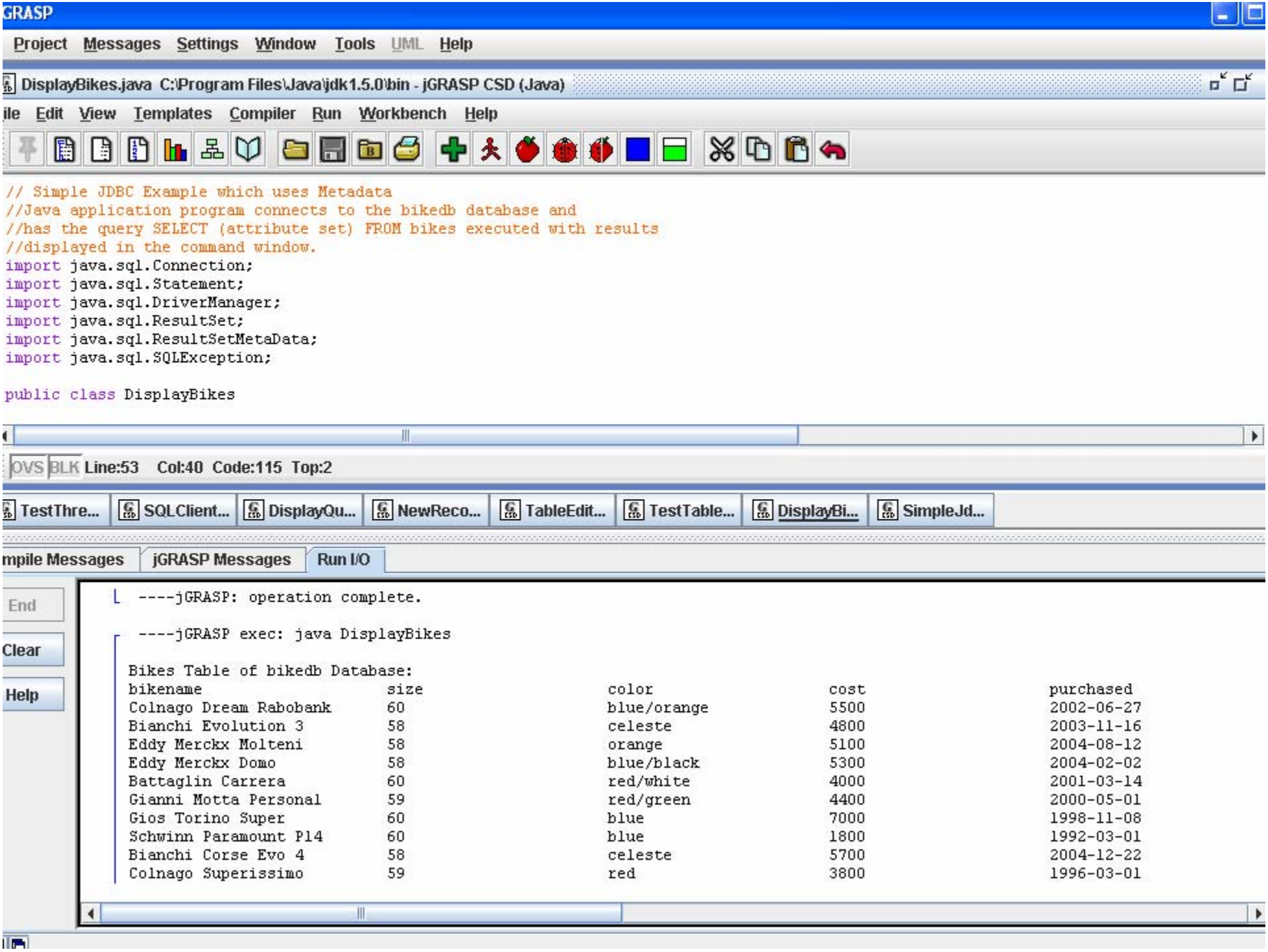
```
// establish connection to database
connection =
   DriverManager.getConnection( DATABASE_URL, "root", “root" );

// create Statement for querying database
statement = connection.createStatement();

// query database
ResultSet resultSet = statement.executeQuery(
   "SELECT bikename, cost, mileage FROM bikes" );

// process query results
ResultSetMetaData metaData = resultSet.getMetaData();
int numberOfColumns = metaData.getColumnCount();
System.out.println( "Bikes Table of bikedb Database:" );

for ( int i = 1; i <= numberOfColumns; i++ )
   System.out.printf( "%-20s\t", metaData.getColumnName( i ) );
System.out.println();

while ( resultSet.next() )
{
   for ( int i = 1; i <= numberOfColumns; i++ )
      System.out.printf( "%-20s\t", resultSet.getObject( i ) );
```

The MySQL query to be executed remotely.

Get metadata from the resultSet to be used for the output formatting.

Simple JDBC Application – page 3

```
            System.out.println();
          } // end while
        }  // end try
        catch ( SQLException sqlException )   {
          sqlException.printStackTrace();
          System.exit( 1 );
        } // end catch
        catch ( ClassNotFoundException classNotFound )  {
          classNotFound.printStackTrace();
          System.exit( 1 );
        } // end catch
        finally  { // ensure statement and connection are closed properly
          try {
               statement.close();
               connection.close();
          } // end try
          catch ( Exception exception ) {
            exception.printStackTrace();
            System.exit( 1 );
          } // end catch
        } // end finally
      } // end main
    }  // end class DisplayBikes
```



GRASP
Project Messages Settings Window Tools UML Help
DisplayBikes.java C:\Program Files\Java\jdk1.5.0\bin - jGRASP CSD (Java)
ile Edit View Templates Compiler Run Workbench Help
// Simple JDBC Example which uses Metadata
//Java application program connects to the bikedb database and
//has the query SELECT (attribute set) FROM bikes executed with results
//displayed in the command window.
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
public class DisplayBikes
OVS BLK Line:53 Col:40 Code:115 Top:2
TestThre... SQLClient... DisplayQu... NewReco... TableEdit... TestTable... DisplayBi... SimpleJd...
mpile Messages jGRASP Messages Run I/O
End
Clear
Help
----jGRASP: operation complete.
----jGRASP exec: java DisplayBikes
Bikes Table of bikedb Database:
bikename size color cost purchased
Colnago Dream Rabobank 60 blue/orange 5500 2002-06-27
Bianchi Evolution 3 58 celeste 4800 2003-11-16
Eddy Merckx Molteni 58 orange 5100 2004-08-12
Eddy Merckx Domo 58 blue/black 5300 2004-02-02
Battaglin Carrera 60 red/white 4000 2001-03-14
Gianni Motta Personal 59 red/green 4400 2000-05-01
Gios Torino Super 60 blue 7000 1998-11-08
Schwinn Paramount P14 60 blue 1800 1992-03-01
Bianchi Corse Evo 4 58 celeste 5700 2004-12-22
Colnago Superissimo 59 red 3800 1996-03-01

Note Regarding Static Method `forName`

- The database driver must be loaded before connecting to the database. The static method `forName` of class `Class` is used to load the class for the database driver.
- This method throws a checked exception of type `java.lang.ClassNotFoundException` if the class loader cannot locate the driver class.
- To avoid this exception, you need to include the `mysql-connector-java-3.1.7-bin.jar` in your program's classpath when you execute the program.
- Copy the `mysql-connector-java-3.1.7-bin.jar` file to the JRE's lib\ext directory. (available on the class web-site, if you didn't already get it from MySQL (www.mysql.com/products/connector/j/)).

Querying the `bikedb` MySQL Database

- In this example, we'll allow the user to enter any valid MySQL query into the Java application to query the bikes table of the bikedb database.
- The results of the query are returned in a JTable, using a TableModel object to provide the ResultSet data to the JTable.
- Class ResultSetTableModel performs the connection to the database and maintains the ResultSet.
- Class DisplayQueryResults creates the GUI and specifies an instance of class ResultSetTableModel to provide the data for the JTable.

Class: ResultSetTableModel – page 1

```
// A TableModel that supplies ResultSet data to a JTable.
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import javax.swing.table.AbstractTableModel;

// ResultSet rows and columns are counted from 1 and JTable
// rows and columns are counted from 0. When processing
// ResultSet rows or columns for use in a JTable, it is
// necessary to add 1 to the row or column number to manipulate
// the appropriate ResultSet column (i.e., JTable column 0 is
// ResultSet column 1 and JTable row 0 is ResultSet row 1).
public class ResultSetTableModel extends AbstractTableModel  {
   private Connection connection;
   private Statement statement;
   private ResultSet resultSet;
   private ResultSetMetaData metaData;
   private int numberOfRows;

   // keep track of database connection status
   private boolean connectedToDatabase = false;
```


Class: ResultSetTableModel – page 2

```
// constructor initializes resultSet and obtains its meta data object;
   // determines number of rows
   public ResultSetTableModel( String driver, String url,
      String username, String password, String query )
      throws SQLException, ClassNotFoundException
   {
      // load database driver class
      Class.forName( driver );

      // connect to database
      connection = DriverManager.getConnection( url, username, password );

      // create Statement to query database
      statement = connection.createStatement(
         ResultSet.TYPE_SCROLL_INSENSITIVE,
         ResultSet.CONCUR_READ_ONLY );

      // update database connection status
      connectedToDatabase = true;

      // set query and execute it
      setQuery( query );
   } // end constructor ResultSetTableModel
```


Class: ResultSetTableModel – page 3

```
// get class that represents column type
public Class getColumnClass( int column ) throws IllegalStateException  {
    // ensure database connection is available
   if ( !connectedToDatabase )
      throw new IllegalStateException( "Not Connected to Database" );

   // determine Java class of column
   try {
      String className = metaData.getColumnClassName( column + 1 );
      // return Class object that represents className
      return Class.forName( className );
   } // end try
   catch ( Exception exception ) {
       exception.printStackTrace();
   } // end catch

   return Object.class; // if problems occur above, assume type Object
} // end method getColumnClass

// get number of columns in ResultSet
public int getColumnCount() throws IllegalStateException  {
    // ensure database connection is available
   if ( !connectedToDatabase )
      throw new IllegalStateException( "Not Connected to Database" );
```


Class: ResultSetTableModel – page 4

```
   // determine number of columns
   try  {
        return metaData.getColumnCount();
   } // end try
   catch ( SQLException sqlException )  {
       sqlException.printStackTrace();
   } // end catch

   return 0; // if problems occur above, return 0 for number of columns
} // end method getColumnCount

// get name of a particular column in ResultSet
public String getColumnName( int column ) throws IllegalStateException  {
    // ensure database connection is available
   if ( !connectedToDatabase )
      throw new IllegalStateException( "Not Connected to Database" );

   // determine column name
   try {
      return metaData.getColumnName( column + 1 );
   } // end try
   catch ( SQLException sqlException )  {
      sqlException.printStackTrace();
   } // end catch
```


Class: ResultSetTableModel – page 5

```
   return ""; // if problems, return empty string for column name
} // end method getColumnName

// return number of rows in ResultSet
public int getRowCount() throws IllegalStateException {
     // ensure database connection is available
   if ( !connectedToDatabase )
      throw new IllegalStateException( "Not Connected to Database" );

   return numberOfRows;
} // end method getRowCount

// obtain value in particular row and column
public Object getValueAt( int row, int column )
   throws IllegalStateException
{
   // ensure database connection is available
   if ( !connectedToDatabase )
      throw new IllegalStateException( "Not Connected to Database" );

   // obtain a value at specified ResultSet row and column
   try  {
      resultSet.absolute( row + 1 );
      return resultSet.getObject( column + 1 );
   } // end try
```


Class: ResultSetTableModel – page 6

```
  catch ( SQLException sqlException ) {
        sqlException.printStackTrace();
   } // end catch

   return ""; // if problems, return empty string object
} // end method getValueAt

// set new database query string
public void setQuery( String query )
   throws SQLException, IllegalStateException
{
   // ensure database connection is available
   if ( !connectedToDatabase )
      throw new IllegalStateException( "Not Connected to Database" );

     // specify query and execute it
   resultSet = statement.executeQuery( query );

   // obtain meta data for ResultSet
   metaData = resultSet.getMetaData();

   // determine number of rows in ResultSet
   resultSet.last();                   // move to last row
   numberOfRows = resultSet.getRow();  // get row number
```


Class: ResultSetTableModel – page 7

```
      // notify JTable that model has changed
      fireTableStructureChanged();
    } // end method setQuery

    // close Statement and Connection
    public void disconnectFromDatabase() {
        if ( !connectedToDatabase )
          return;

      // close Statement and Connection
      try  {
        statement.close();
        connection.close();
      } // end try
      catch ( SQLException sqlException ) {
        sqlException.printStackTrace();
      } // end catch
      finally  // update database connection status
      {
        connectedToDatabase = false;
      } // end finally
    } // end method disconnectFromDatabase
  }  // end class ResultSetTableModel
```


Class: DisplayQueryResults – page 1

```
// Display the contents of the bikes table in the bikedb database.
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.sql.SQLException;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JScrollPane;
import javax.swing.ScrollPaneConstants;
import javax.swing.JTable;
import javax.swing.JOptionPane;
import javax.swing.JButton;
import javax.swing.Box;

public class DisplayQueryResults extends JFrame
{
   // JDBC driver, database URL, username and password
   static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
   static final String DATABASE_URL = "jdbc:mysql://localhost/bikedb";
   static final String USERNAME= "root";
   static final String PASSWORD= “root";
```


Class: DisplayQueryResults – page 2

```
 // default query retrieves all data from bikes table
static final String DEFAULT_QUERY = "SELECT * FROM bikes";

private ResultSetTableModel tableModel;
private JTextArea queryArea;

// create ResultSetTableModel and GUI
public DisplayQueryResults()  {
     super( "Displaying Query Results" );
     // create ResultSetTableModel and display database table
   try {
      // create TableModel for results of query SELECT * FROM bikes
      tableModel = new ResultSetTableModel( JDBC_DRIVER, DATABASE_URL,
         USERNAME, PASSWORD, DEFAULT_QUERY );

      // set up JTextArea in which user types queries
      queryArea = new JTextArea( DEFAULT_QUERY, 3, 100 );
      queryArea.setWrapStyleWord( true );
      queryArea.setLineWrap( true );

      JScrollPane scrollPane = new JScrollPane( queryArea,
         ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
         ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER );
```


Class: DisplayQueryResults – page 3

```
// set up JButton for submitting queries
JButton submitButton = new JButton( "Submit Query" );

// create Box to manage placement of queryArea and
// submitButton in GUI
Box box = Box.createHorizontalBox();
box.add( scrollPane );
box.add( submitButton );

// create JTable delegate for tableModel
JTable resultTable = new JTable( tableModel );

// place GUI components on content pane
add( box, BorderLayout.NORTH );
add( new JScrollPane( resultTable ), BorderLayout.CENTER );

// create event listener for submitButton
submitButton.addActionListener(

   new ActionListener()  {
      // pass query to table model
      public void actionPerformed( ActionEvent event )  {
         // perform a new query
```


Class: DisplayQueryResults – page 4

```
            try  {
               tableModel.setQuery( queryArea.getText() );
            } // end try
            catch ( SQLException sqlException )  {
                  JOptionPane.showMessageDialog( null,
                  sqlException.getMessage(), "Database error",
                  JOptionPane.ERROR_MESSAGE );
               // try to recover from invalid user query by executing default query
              try   {
                 tableModel.setQuery( DEFAULT_QUERY );
                 queryArea.setText( DEFAULT_QUERY );
              } // end try
              catch ( SQLException sqlException2 )  {
                    JOptionPane.showMessageDialog( null,
                    sqlException2.getMessage(), "Database error",
                    JOptionPane.ERROR_MESSAGE );
                    // ensure database connection is closed
                 tableModel.disconnectFromDatabase();
                System.exit( 1 ); // terminate application
              } // end inner catch
            } // end outer catch
          } // end actionPerformed
        }  // end ActionListener inner class
      ); // end call to addActionListener
```

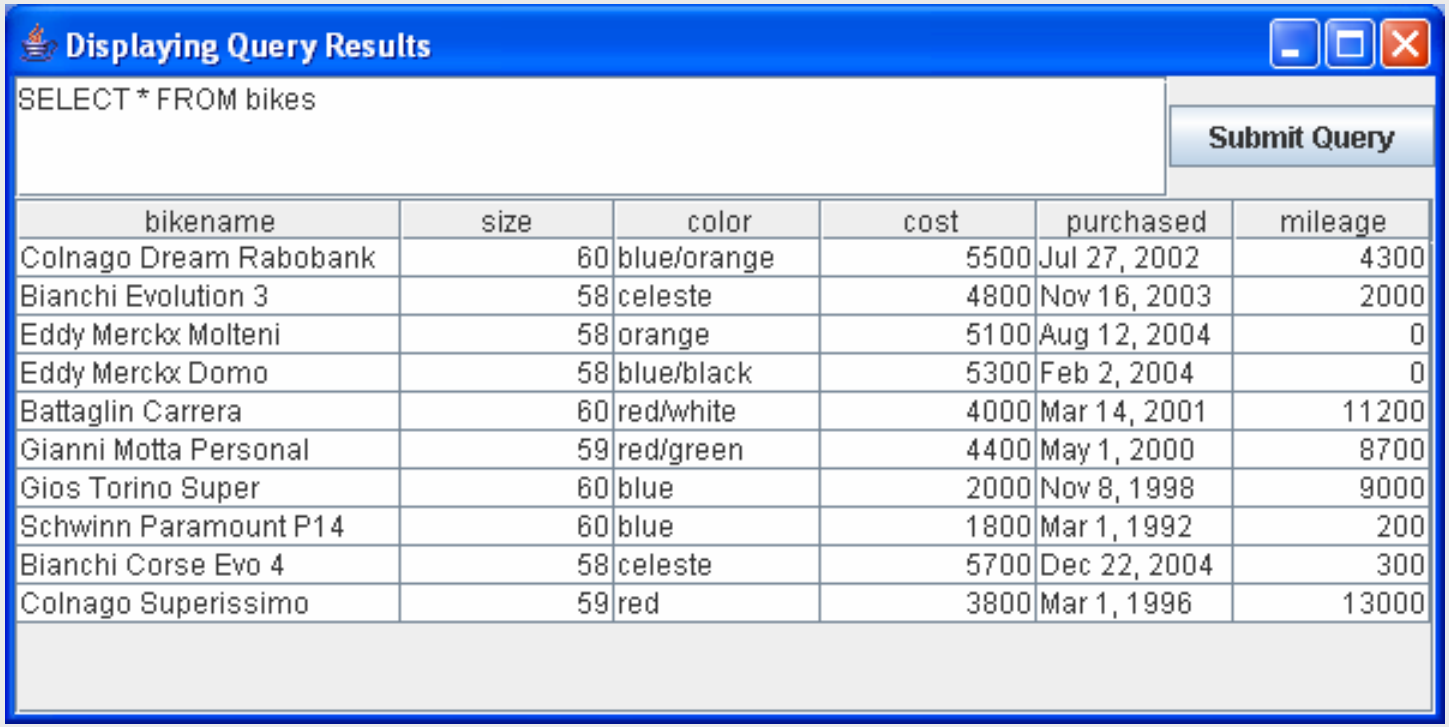

Class: DisplayQueryResults – page 5

```
   setSize( 500, 250 ); // set window size
   setVisible( true ); // display window
} // end try
catch ( ClassNotFoundException classNotFound )  {
      JOptionPane.showMessageDialog( null,
      "MySQL driver not found", "Driver not found",
      JOptionPane.ERROR_MESSAGE );

   System.exit( 1 ); // terminate application
} // end catch
catch ( SQLException sqlException )  {
      JOptionPane.showMessageDialog( null, sqlException.getMessage(),
      "Database error", JOptionPane.ERROR_MESSAGE );
   // ensure database connection is closed
   tableModel.disconnectFromDatabase();
  System.exit( 1 );   // terminate application
} // end catch
 // dispose of window when user quits application (this overrides
// the default of HIDE_ON_CLOSE)
setDefaultCloseOperation( DISPOSE_ON_CLOSE );
 // ensure database connection is closed when user quits application
addWindowListener(
```


Class: DisplayQueryResults – page 6

```
            new WindowAdapter()
            {
               // disconnect from database and exit when window has closed
               public void windowClosed( WindowEvent event )
               {
                  tableModel.disconnectFromDatabase();
                  System.exit( 0 );
               } // end method windowClosed
            } // end WindowAdapter inner class
         ); // end call to addWindowListener
      } // end DisplayQueryResults constructor

      // execute application
      public static void main( String args[] )
      {
         new DisplayQueryResults();
      } // end main
   } // end class DisplayQueryResults
```

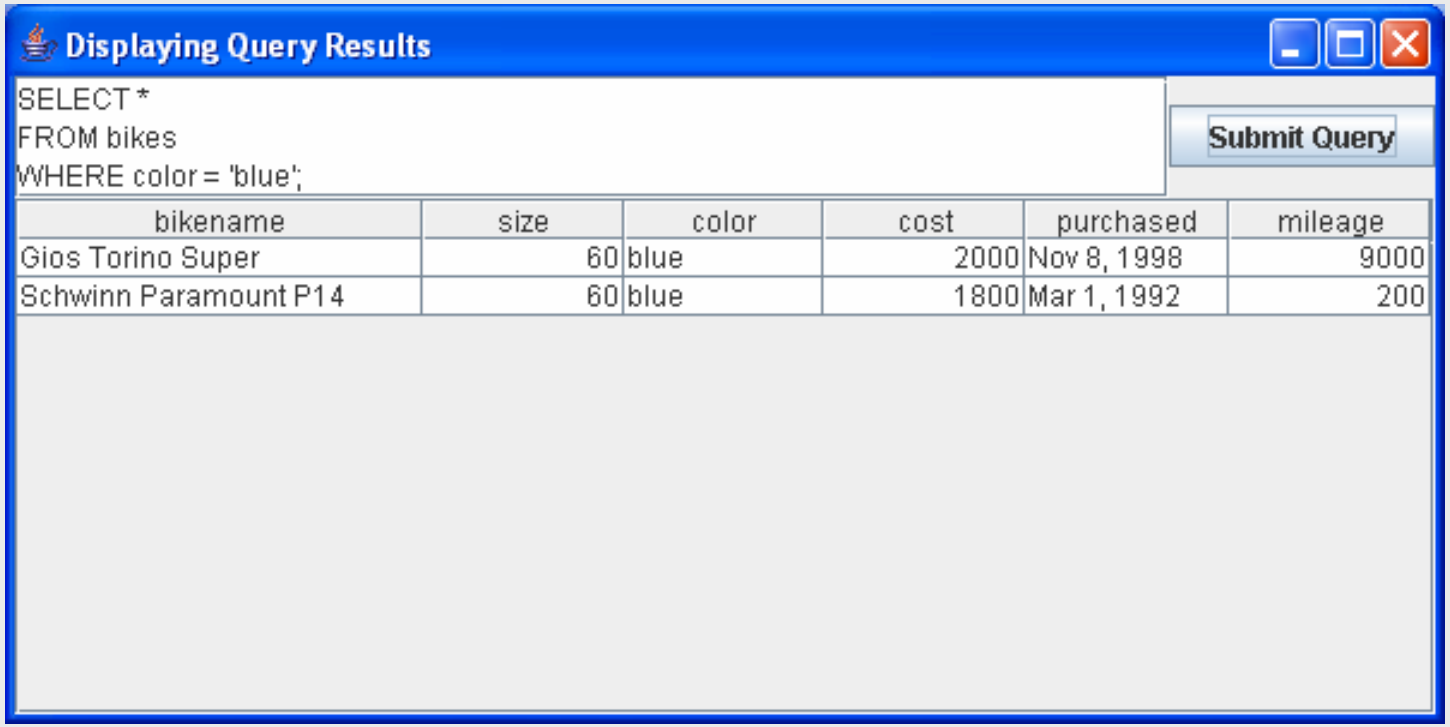

Execution of DisplayQueryResults



bikename	size	color	cost	purchased	mileage
Colnago Dream Rabobank	60	blue/orange	5500	Jul 27, 2002	4300
Bianchi Evolution 3	58	celeste	4800	Nov 16, 2003	2000
Eddy Merckx Molteni	58	orange	5100	Aug 12, 2004	0
Eddy Merckx Domo	58	blue/black	5300	Feb 2, 2004	0
Battaglin Carrera	60	red/white	4000	Mar 14, 2001	11200
Gianni Motta Personal	59	red/green	4400	May 1, 2000	8700
Gios Torino Super	60	blue	2000	Nov 8, 1998	9000
Schwinn Paramount P14	60	blue	1800	Mar 1, 1992	200
Bianchi Corse Evo 4	58	celeste	5700	Dec 22, 2004	300
Colnago Superissimo	59	red	3800	Mar 1, 1996	13000

Display of default query results from DisplayQueryResults application

Execution of DisplayQueryResults



Display of user-formed query results from DisplayQueryResults application