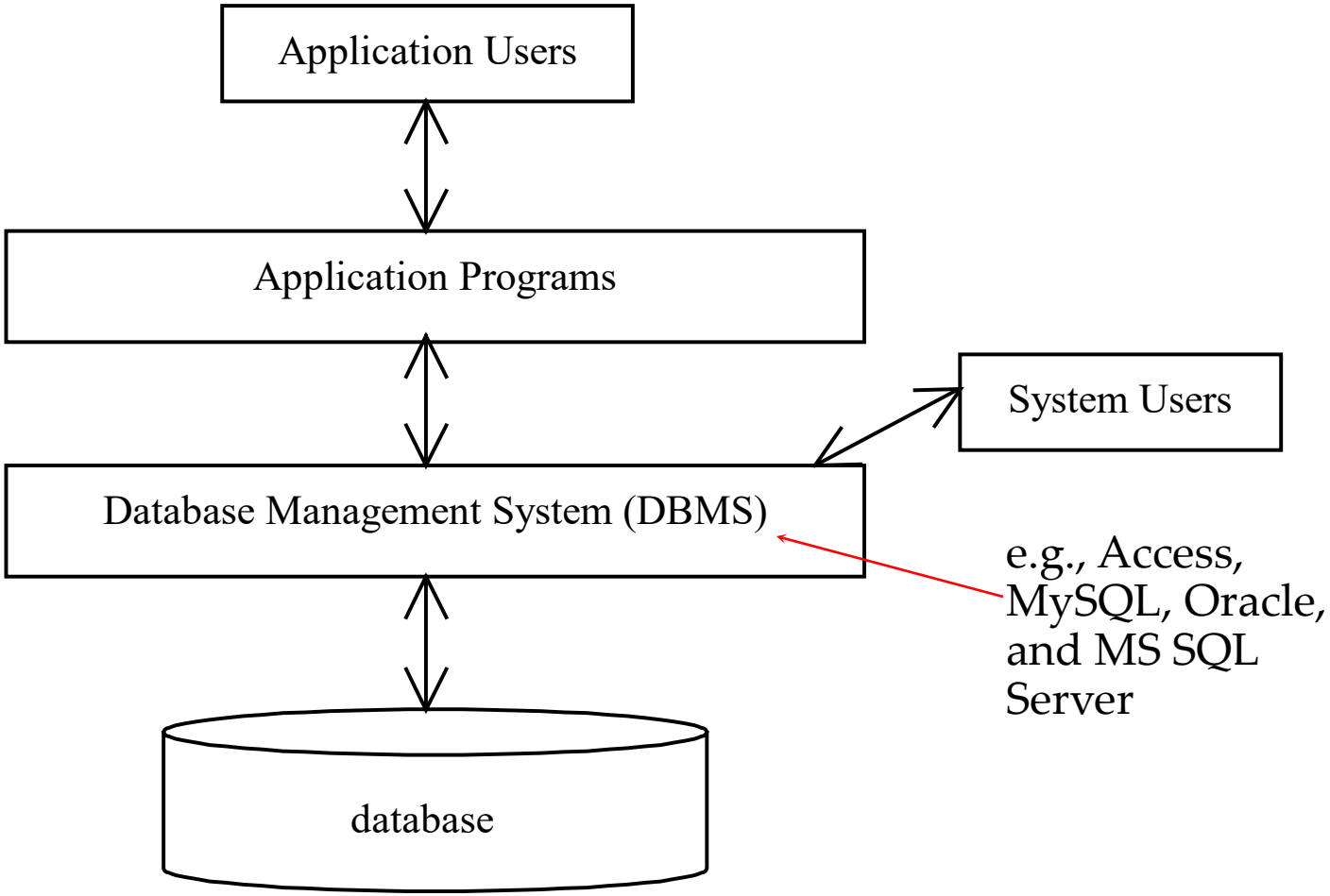
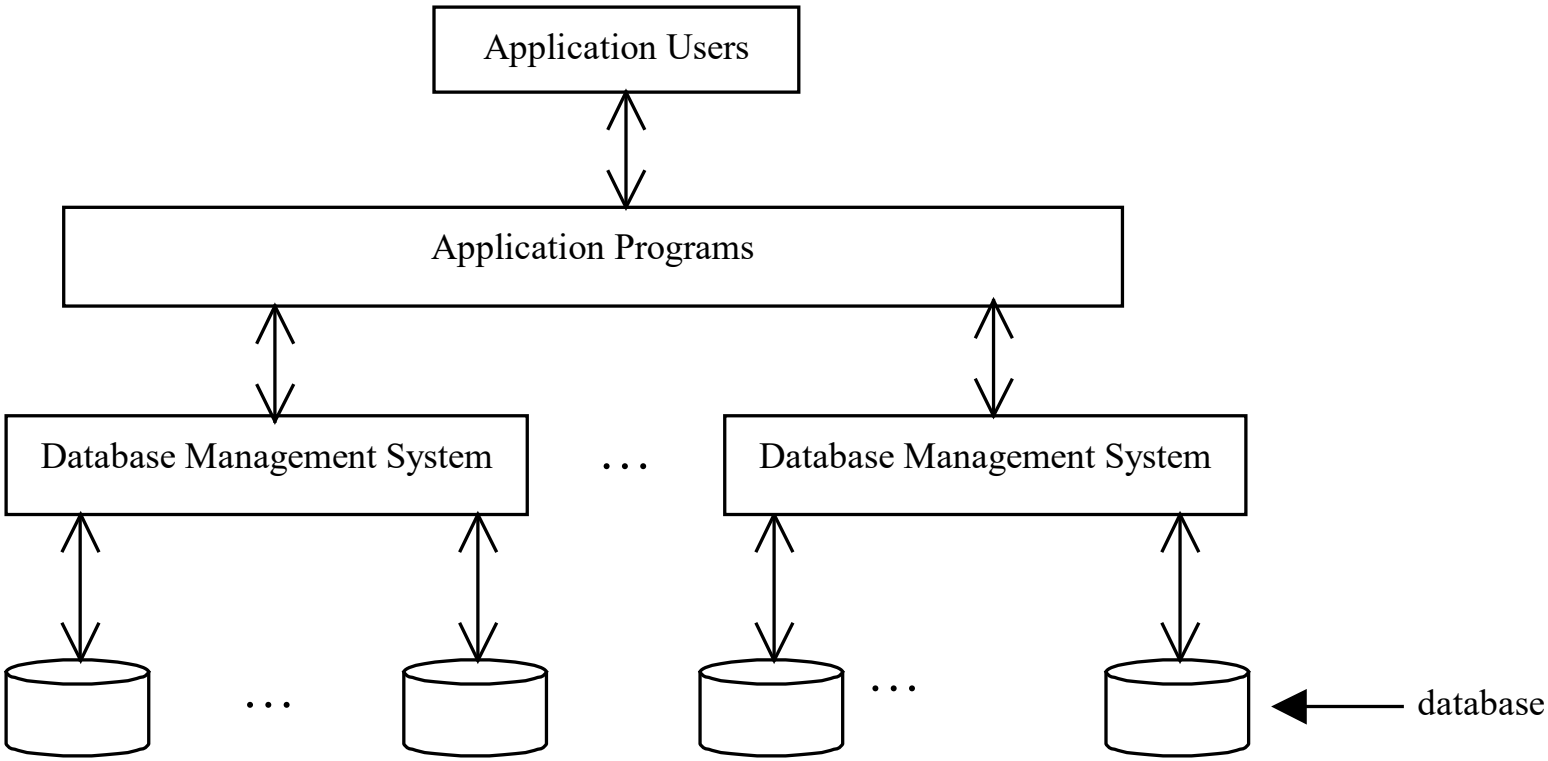
27 Java Database Programming

Chapter 37@8e

What is a Database System?



Database Application Systems



Rational Database and Relational Data Model

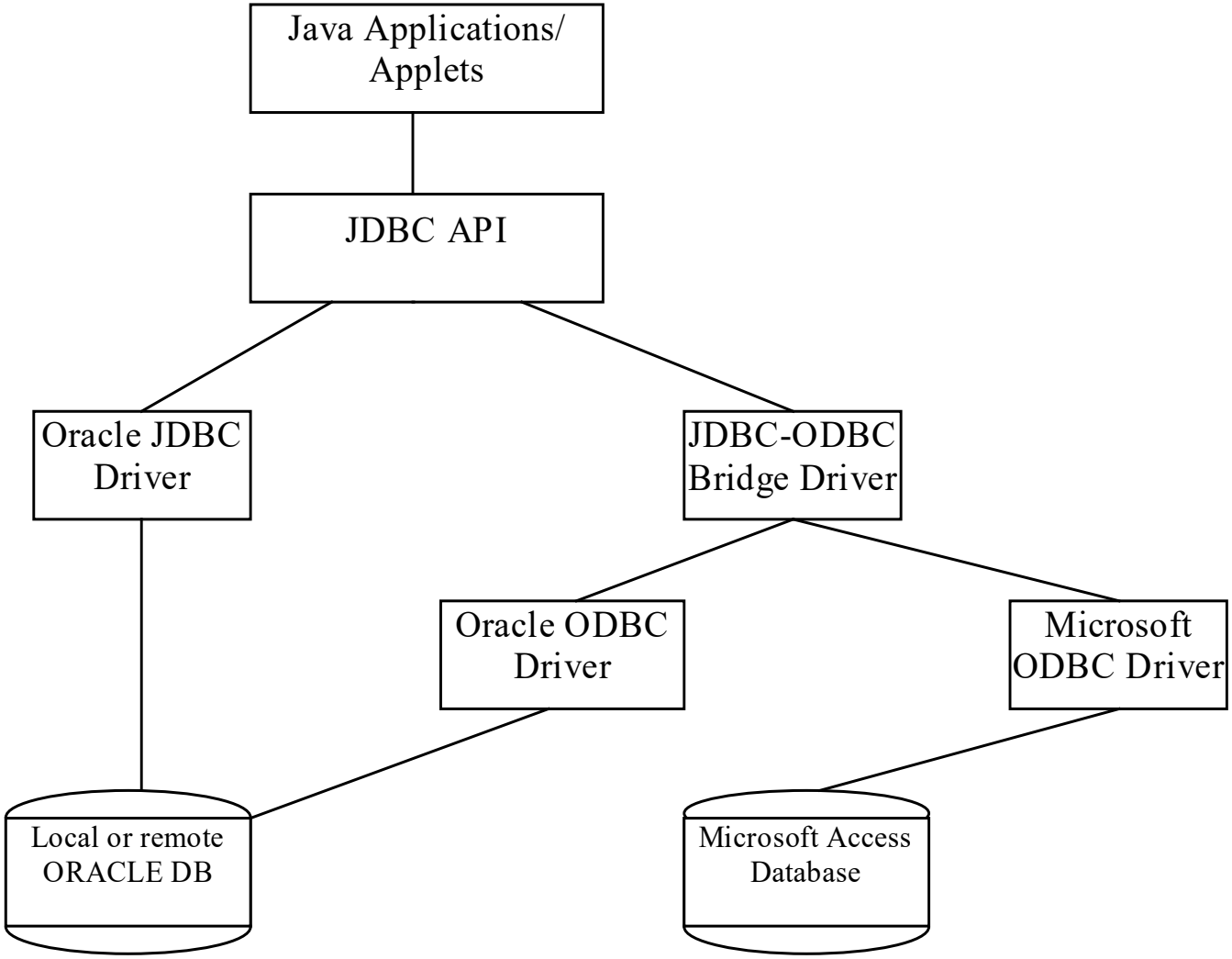
Most of today's database systems are relational database systems, based on the relational data model. A relational data model has three key components: structure, integrity and languages.

- *Structure* defines the representation of the data.
- *Integrity* imposes constraints on the data.
- *Language* provides the means for accessing and manipulating data.

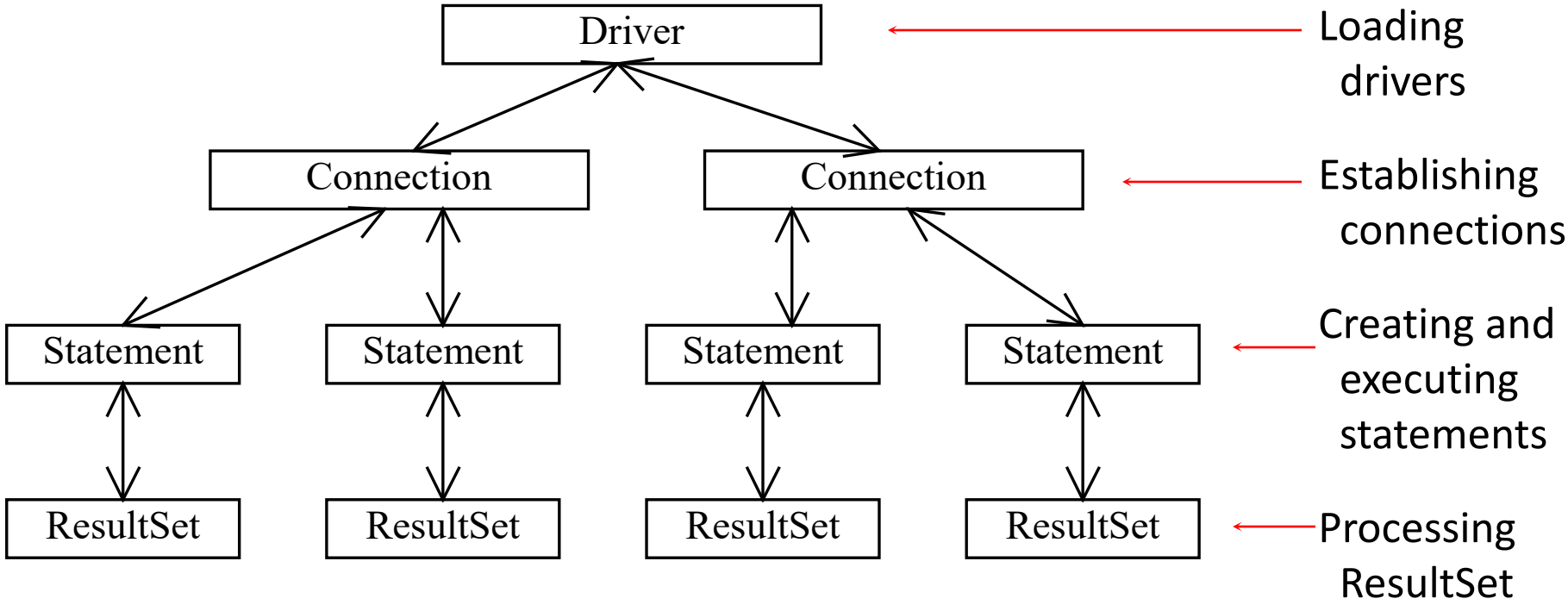
Why Java for Database Programming?

- First, Java is platform independent. You can develop platform-independent database applications using SQL and Java for any relational database systems.
- Second, the support for accessing database systems from Java is built into Java API, so you can create database applications using all Java code with a common interface.
- Third, Java is taught in almost every university either as the first programming language or as the second programming language.

The Architecture of JDBC



The JDBC Interfaces



Developing JDBC Programs

Loading drivers

Establishing connections

Creating and executing statements

Processing ResultSet

Statement to load a driver:

```
Class.forName("JDBCDriverClass");
```

A driver is a class. For example:

Database	Driver Class	Source
Access	sun.jdbc.odbc.JdbcOdbcDriver	Already in JDK
MySQL	com.mysql.jdbc.Driver	Website
Oracle	oracle.jdbc.driver.OracleDriver	Website

The JDBC-ODBC driver for Access is bundled in JDK.

MySQL driver class is in mysqljdbc.jar

Oracle driver class is in classes12.jar

To use the MySQL and Oracle drivers, you have to add mysqljdbc.jar and classes12.jar in the classpath using the following DOS command on Windows:

```
classpath=%classpath%;c:\book\mysqljdbc.jar;c:\book\classes12.jar
```

Developing JDBC Programs

Loading drivers

Establishing connections

Creating and executing statements

Processing ResultSet

```
Connection connection = DriverManager.getConnection(databaseURL);
```

Database	URL Pattern
Access	jdbc:odbc:dataSource
MySQL	jdbc:mysql://hostname/dbname
Oracle	jdbc:oracle:thin:@hostname:port#:oracleDBSID

Examples:

For Access:

```
Connection connection = DriverManager.getConnection
  ("jdbc:odbc:ExampleMDBDataSource");
```

See Supplement IV.D for creating an ODBC data source

For MySQL:

```
Connection connection = DriverManager.getConnection
  ("jdbc:mysql://localhost/test");
```

For Oracle:

```
Connection connection = DriverManager.getConnection
  ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",  "scott", "tiger");
```

Developing JDBC Programs

Loading drivers

Establishing connections

Creating and executing statements

Processing ResultSet

Creating statement:

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

Executing statement (for update, delete, insert):

```
statement.executeUpdate
    ("create table Temp (col1 char(5), col2 char(5))");
```

Executing statement (for select):

```
// Select the columns from the Student table
ResultSet resultSet = statement.executeQuery
  ("select firstName, mi, lastName from Student where lastName "
    + " = 'Smith'");
```

Developing JDBC Programs

Loading drivers

Establishing connections

Creating and executing statements

Processing ResultSet

Executing statement (for select):

```
// Select the columns from the Student table
ResultSet resultSet = stmt.executeQuery
  ("select firstName, mi, lastName from Student where lastName "
    + " = 'Smith'");
```

Processing ResultSet (for select):

```
// Iterate through the result and print the student names
while (resultSet.next())
  System.out.println(resultSet.getString(1) + " " + resultSet.getString(2)
    + ". " + resultSet.getString(3));
```

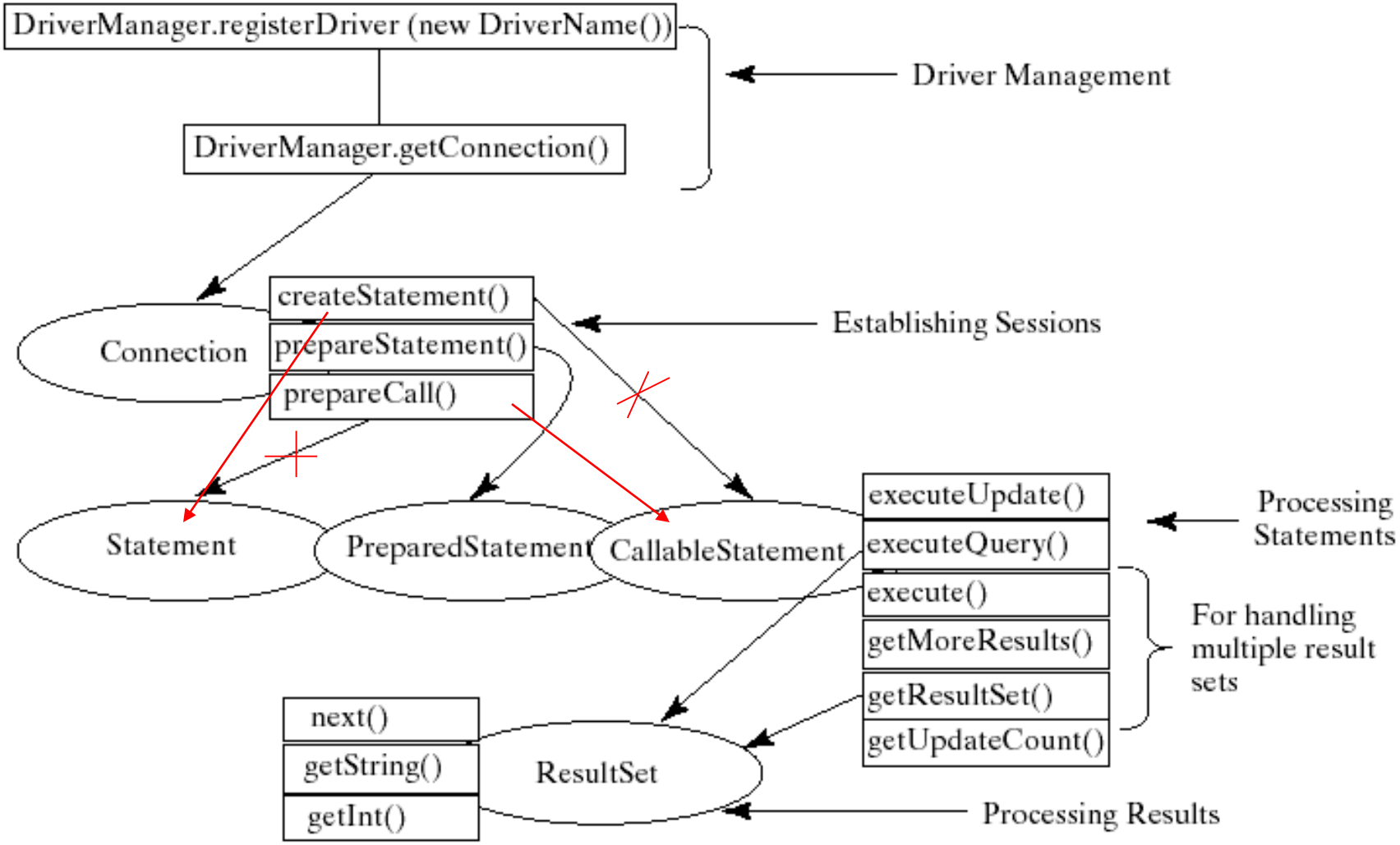
```java
import java.sql.*;
public class SimpleJdbc {
  public static void main(String[] args)
      throws SQLException, ClassNotFoundException {
    // Load the JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");

    // Establish a connection
    Connection connection = DriverManager.getConnection
      ("jdbc:mysql://localhost/test");
    System.out.println("Database connected");

    // Create a statement
    Statement statement = connection.createStatement();

    // Execute a statement
    ResultSet resultSet = statement.executeQuery
      ("select firstName, mi, lastName from Student where lastName "
        + " = 'Smith'");

    // Iterate through the result and print the student names
    while (resultSet.next())
      System.out.println(resultSet.getString(1) + "\t" +
        resultSet.getString(2) + "\t" + resultSet.getString(3));

    // Close the connection
    connection.close();
  }
}
```

Processing Statements

Once a connection to a particular database is established, it can be used to send SQL statements from your program to the database.

JDBC provides the Statement, PreparedStatement, and CallableStatement interfaces to **facilitate** sending statements to a database for execution and receiving execution results from the database.

Processing Statements Diagram



The execute, executeQuery, and executeUpdate Methods

The methods for executing SQL statements are execute, executeQuery, and executeUpdate, each of which accepts a string containing a SQL statement as an argument. This string is passed to the database for execution.

The execute method should be used if the execution produces multiple result sets, multiple update counts, or a combination of result sets and update counts.

The execute, executeQuery, and executeUpdate Methods, cont.

The executeQuery method should be used if the execution produces a single result set, such as the SQL select statement.

The executeUpdate method should be used if the statement results in a single update count or no update count, such as a SQL INSERT, DELETE, UPDATE, or DDL statement.

事务处理

- 在数据库操作中，一项事务是指由一条或多条对数据库更新的sql语句所组成的一个不可分割的工作单元。只有当事务中的所有操作都正常完成了，整个事务才能被提交到数据库，如果有一项操作没有完成，就必须撤消整个事务。

```
public int delete(int sID) {
    dbc = new DataBaseConnection();
    Connection con = dbc.getConnection();
    try {
        con.setAutoCommit(false);// 更改JDBC事务的默认提交方式
        dbc.executeUpdate("delete from xiao where ID=" + sID);
        dbc.executeUpdate("delete from xiao_content where ID=" + sID);
        dbc.executeUpdate("delete from xiao_affix where bylawid=" + sID);
        con.commit();//提交JDBC事务
        con.setAutoCommit(true);// 恢复JDBC事务的默认提交方式
        dbc.close();
        return 1;
    }
    catch (Exception exc) {
        con.rollBack();//回滚JDBC事务
        exc.printStackTrace();
        dbc.close();
        return -1;
    }
}
```

事务处理

在connection类中提供了3个控制事务的方法：

（1） setAutoCommit(Boolean autoCommit):设置是否自动提交事务；

（2） commit();提交事务；

（3） rollback();撤消事务；

- 在jdbc api中，**默认的情况为自动提交事务**，也就是说，**每一条对数据库的更新的sql语句代表一项事务**，操作成功后，系统自动调用commit来提交，否则将调用rollback来撤消事务。

  在jdbc api中，可以通过调用setAutoCommit(false) 来禁止自动提交事务。然后就可以把多条更新数据库的sql语句做为一个事务，在所有操作完成之后，调用commit来进行整体提交。倘若其中一项 sql操作失败，就不会执行commit方法，而是产生相应的sqlexception，此时就可以捕获异常代码块中调用rollback方法撤消事务。

PreparedStatement

The PreparedStatement interface is designed to execute dynamic SQL statements and SQL-stored procedures with IN parameters. These SQL statements and stored procedures are **precompiled** for efficient use when repeatedly executed.

```
Statement pstmt = connection.prepareStatement
  ("insert into Student (firstName, mi, lastName) +
   values (?, ?, ?)");
```

```
String queryString = "select firstName, mi, " +
  "lastName, title, grade from Student, Enrollment, Course " +
  "where Student.ssn = ? and Enrollment.courseId = ? " +
  "and Enrollment.courseId = Course.courseId";

// Create a statement
preparedStatement = connection.prepareStatement(queryString);
```

```
preparedStatement.setString(1, ssn);
preparedStatement.setString(2, courseId);
ResultSet rset = preparedStatement.executeQuery();

if (rset.next()) {
  String lastName = rset.getString(1);
  String mi = rset.getString(2);
  String firstName = rset.getString(3);
  String title = rset.getString(4);
  String grade = rset.getString(5);
```

Retrieving Database Metadata

Database metadata is the information that describes database itself.

JDBC provides the DatabaseMetaData interface for obtaining database wide information and the ResultSetMetaData interface for obtaining the information on the specific ResultSet.

DatabaseMetadata, cont.

The DatabaseMetaData interface provides more than 100 methods for getting database metadata concerning the database as a whole.

These methods can be divided into three groups: for retrieving general information, for finding database capabilities, and for getting object descriptions.

General Information

The general information includes the URL, username, product name, product version, driver name, driver version, available functions, available data types and so on.

Obtaining Database Capabilities

The examples of the database capabilities are whether the database supports the GROUP BY operator, the ALTER TABLE command with add column option, supports entry-level or full ANSI92 SQL grammar.

Obtaining Object Descriptions

<u>the examples of the database objects are tables, views, and procedures</u>.

```
DatabaseMetaData dbMetaData = connection.getMetaData();

System.out.println("database URL: " + dbMetaData.getURL());
System.out.println("database username: " +
  dbMetaData.getUserName());
System.out.println("database product name: " +
  dbMetaData.getDatabaseProductName());
System.out.println("database product version: " +
  dbMetaData.getDatabaseProductVersion());
System.out.println("JDBC driver name: " +
  dbMetaData.getDriverName());
System.out.println("JDBC driver version: " +
  dbMetaData.getDriverVersion());
System.out.println("JDBC driver major version: " +
  new Integer(dbMetaData.getDriverMajorVersion()));
System.out.println("JDBC driver minor version: " +
  new Integer(dbMetaData.getDriverMinorVersion()));
System.out.println("Max number of connections: " +
  new Integer(dbMetaData.getMaxConnections()));
System.out.println("MaxTableNameLentgh: " +
  new Integer(dbMetaData.getMaxTableNameLength()));
System.out.println("MaxColumnsInTable: " +
  new Integer(dbMetaData.getMaxColumnsInTable()));
connection.close();
```

Sample run on next slide

Sample Run

Command Prompt

```
database URL: jdbc:mysql://localhost/test
database username: nobody@localhost
database product name: MySQL
database product version: 4.0.14-max-debug
JDBC driver name: MySQL-AB JDBC Driver
JDBC driver version: mysql-connector-java-3.0.9-stable
:12 $, $Revision: 1.27.2.25 $ >
JDBC driver major version: 3
JDBC driver minor version: 0
Max number of connections: 0
MaxTableNameLentgh: 64
MaxColumnsInTable: 512
```

Batch Updates

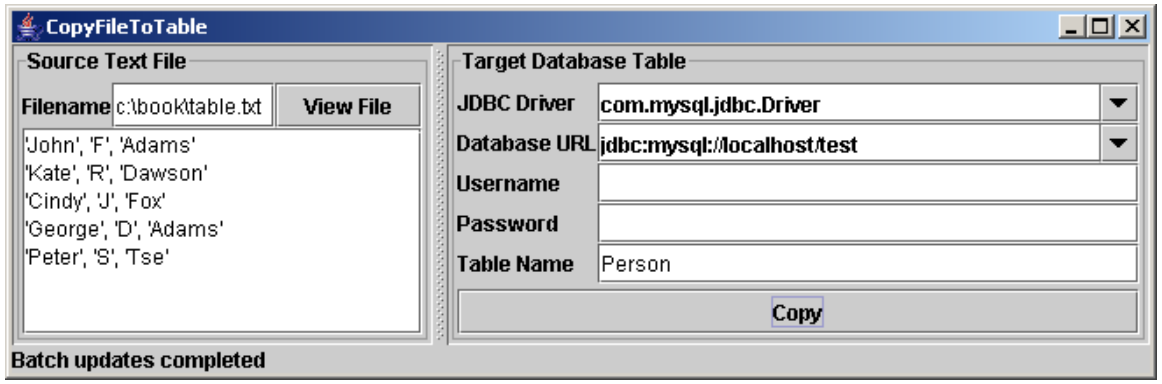
To improve performance, JDBC 2 introduced the batch update for processing nonselect SQL commands. A batch update consists of a sequence of **nonselect** SQL commands. These commands are collected in a batch and submitted to the database all together.

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

// Add SQL commands to the batch
statement.addBatch("create table T (C1 integer, C2 varchar(15))");
statement.addBatch("insert into T values (100, 'Smith')");
statement.addBatch("insert into T values (200, 'Jones')");

// Execute the batch
int count[] = statement.executeBatch();
```

The executeBatch() method returns an array of counts, each of which counts the number of the rows affected by the SQL command. The first count returns 0 because it is a DDL command. The rest of the commands return 1 because only one row is affected.

Example: Copying Text Files to Table

Write a program that gets data from a text file and copies the data to a table. The text file consists of the lines, each of which corresponds to a row in the table. The fields in a row are separated by commas. The string values in a row are enclosed in single quotes. You can view the text file by clicking the View File button and copy the text to the table by clicking the Copy button. The table must already be defined in the database.



CopyFileToTable

Run

```java
// Determine if batchUpdatesSupported is supported
boolean batchUpdatesSupported = false;

try {
  if (connection.getMetaData().supportsBatchUpdates()) {
    batchUpdatesSupported = true;
    System.out.println("batch updates supported");
  }
  else {
    System.out.println("The driver " +
      "does not support batch updates");
  }
}
catch (UnsupportedOperationException ex) {
  System.out.println("The operation is not supported");
}
```

```java
// Build the SQL INSERT statement
String sqlInsert = "insert into " + tfTa
  + " values (";

                    ..........

// Determine if the driver is capa
if (batchUpdatesSupported) {
  // Read a line and add the inser
  while (input.hasNext()) {
    statement.addBatch(sqlInsert + input.nextLine() + ")");
  }

  statement.executeBatch();

  lblStatus.setText("Batch updates completed");
}
else {
  // Read a line and execute insert table command
  while (input.hasNext()) {
    statement.executeUpdate(sqlInsert + input.nextLine() + ")");
  }

  lblStatus.setText("Single row update completed");
}
```

Scrollable and Updateable Result Set

The result sets used in the preceding examples are read sequentially. A result set maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The next() method moves the cursor forward to the next row. This is known as *sequential forward reading*. It is the only way of processing the rows in a result set that is supported by JDBC 1.

With JDBC 2, you can scroll the rows both forward and backward and move the cursor to a desired location using the first, last, next, previous, absolute, or relative method. Additionally, you can insert, delete, or update a row in the result set and have the changes automatically reflected in the database.

Creating Scrollable Statements

To obtain a scrollable or updateable result set, you must first create a statement with an appropriate type and concurrency mode. For a static statement, use

```
Statement statement = connection.createStatement
  (int resultSetType, int resultSetConcurrency);
```

For a prepared statement, use

```
PreparedStatement statement = connection.prepareStatement
  (String sql, int resultSetType, int resultSetConcurrency);
```

The resulting set is scrollable

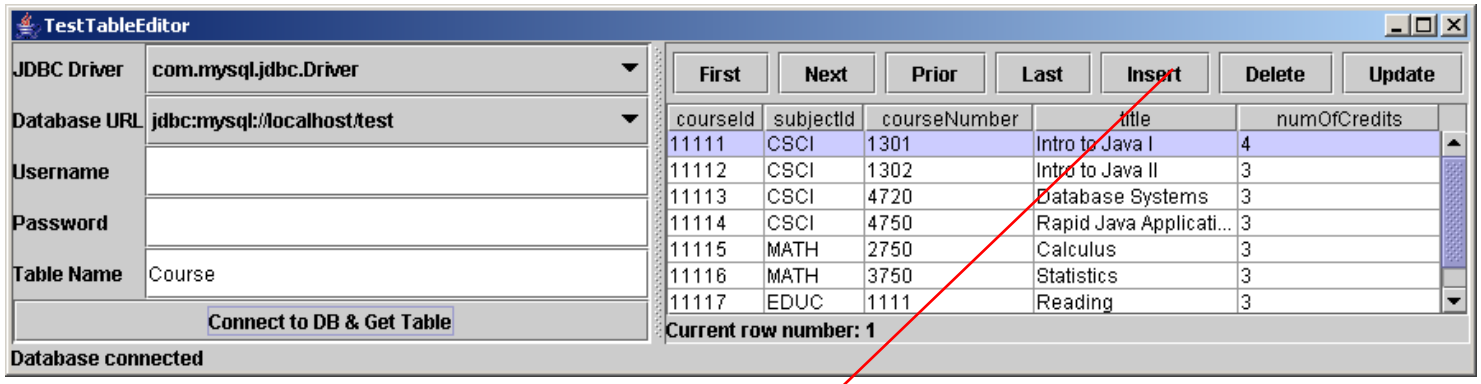
```
ResultSet resultSet = statement.executeQuery(query);
```

TYPE_FORWARD_ONLY
TYPE_SCROLL_INSENSITIVE
TYPE_SCROLL_SENSITIVE

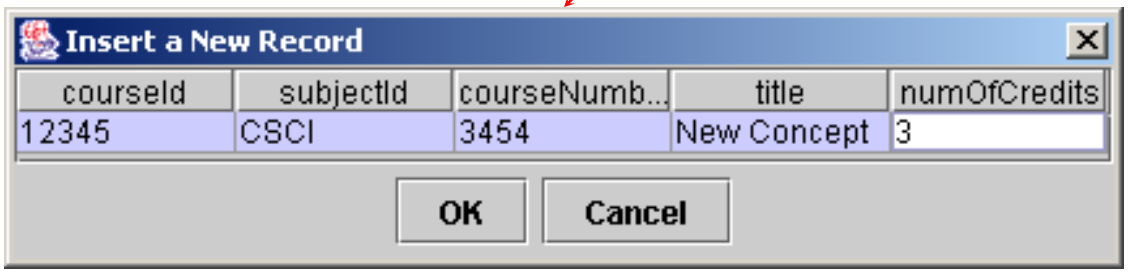
CONCUR_READ_ONLY
CONCUR_UPDATABLE

Example: Scrolling and Updating Table

Develop a useful utility that displays all the rows of a database table in a JTable and uses a scrollable and updateable result set to navigate the table and modify its contents. defined in the database.



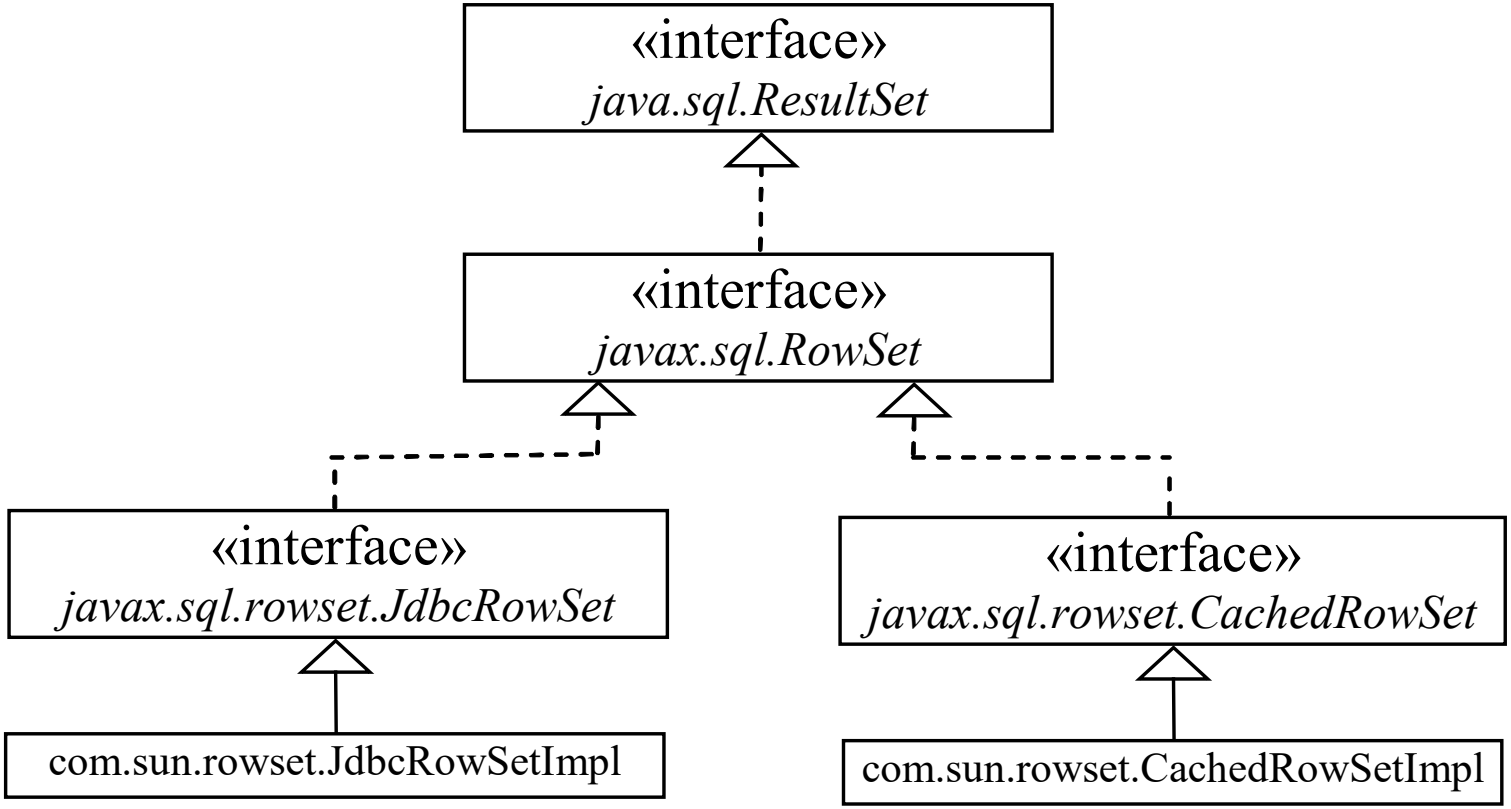
Insert a new row



TestTableEditor | TableEditor | NewRecordDialog | Run

RowSet: JdbcRowSet and CachedRowSet

JDBC 2 introduced a new RowSet interface that can be used to simplify database programming. The RowSet interface extends java.sql.ResultSet with additional capabilities that allow a RowSet instance to be configured to connect to a JDBC url, username, password, set a SQL command, execute the command, and retrieve the execution result.



- JdbcRowSet是连接式(Connected)的RowSet，也就是操作JdbcRowSet期间，会保持与数据库的连接，可视为取得、操作ResultSet的行为封装，可简化JDBC程序的编写，或作为JavaBean使用。

```
JdbcRowSet rowset = new JdbcRowSetImpl();
rowset.setUrl("jdbc:mysql://localhost:3306/demo");
rowset.setUsername("root");
rowset.setPassword("123456");
rowset.setCommand("SELECT * FROM t_messages WHERE id = ?");
rowset.setInt(1, 1);
rowset.execute();
```

- CachedRowSet则为离线式(Disconnected)的RowSet(其子接口当然也是)，在查询并填充完数据后，就会断开与数据源的连接，而不用占据相关连接资源，必要时也可以再与数据源连接进行数据同步。

```java
public CachedRowSet query(String sql) throws ClassNotFoundException, SQLException {
    Class.forName(driver);
    Connection conn = DriverManager.getConnection(url, user, pass);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql);

    RowSetFactory factory = RowSetProvider.newFactory();
    CachedRowSet crs = factory.createCachedRowSet();
    crs.populate(rs);

    // 显式关闭所有连接资源
    rs.close();
    stmt.close();
    conn.close();

    return crs; // 这样返回的RowSet仍然能用，说明被离线缓存了
}
```

```java
CachedRowSet rs = query("select * from student_table"); // 虽然连接已关闭，但结果集被离线缓存下来了

rs.afterLast(); // 可滚动
while (rs.previous()) {
    System.out.println( // 解析
        rs.getString(1) + '\t' +
        rs.getString(2) + '\t' +
        rs.getString(3)
    );

    if (rs.getInt("student_id") == 3) { // 更新
        rs.updateString("student_name", "QQQ");
        rs.updateRow();
    }
}

// 重连，同步到真实数据库
Connection conn = DriverManager.getConnection(url, user, pass);
conn.setAutoCommit(false); // 先不管，这个跟事务管理有关
rs.acceptChanges(conn); // 由于之前连接资源已断，因此要调用有参版本的
```

SQL BLOB and CLOB Types

Database can store not only numbers and strings, but also images. SQL3 introduced a new data type BLOB (*B*inary *L*arge *OB*ject) for storing binary data, which can be used to store images.

BLOB

CLOB

Another new SQL3 type is CLOB (*C*haracter *L*arge *OB*ject) for storing a large text in the character format. JDBC 2 introduced the interfaces java.sql.Blob and java.sql.Clob to support mapping for these new SQL types. JBDC 2 also added new methods, such as getBlob, setBinaryStream, getClob, setBlob, and setClob, in the interfaces ResultSet and PreparedStatement to access SQL BLOB, and CLOB values.

To store an image into a cell in a table, the corresponding column for the cell must be of the BLOB type. For example, the following SQL statement creates a table whose type for the flag column is BLOB.

```
create table Country(name varchar(30), flag blob,
  description varchar(255));
```

Storing and Retrieving Images in JDBC

To insert a record with images to a table, define a prepared statement like this one:

```
PreparedStatement pstmt = connection.prepareStatement(
  "insert into Country values(?, ?, ?)");
```

Images are usually stored in files. You may first get an instance of InputStream for an image file and then use the setBinaryStream method to associate the input stream with a cell in the table, as follows:

Store image

```
// Store image to the table cell
File file = new File(imageFilenames[i]);
InputStream inputImage = new FileInputStream(file);
pstmt.setBinaryStream(2, inputImage, (int)(file.length()));
```

Retrieve image

To retrieve an image from a table, use the getBlob method, as shown below:

```
// Store image to the table cell
Blob blob = rs.getBlob(2);     //or Blob blob = rs.getBlob("flag");
ImageIcon imageIcon = new ImageIcon(
  blob.getBytes(1, (int)blob.length()));   //getBytes(pos, length);
```

Example: Scrolling and Updating Table

In this example, you will create a table, populate it with data, including images, and retrieve and display images. The table is named Country. Each record in the table consists of three fields: name, flag, and description. Flag is an image field. The program first creates the table and stores data to it. Then the program retrieves the country names from the table and adds them to a combo box. When the user selects a name from the combo box, the country's flag and description are displayed.

StoreAndRetrieveImage

Run

```java
private void storeDataToTable() {
  String[] countries = {"Canada", "UK", "USA", "Germany",
    "Indian", "China"};

  String[] imageFilenames = {"image/ca.gif", "image/uk.gif",
    "image/us.gif", "image/germany.gif", "image/india.gif",
    "image/china.gif"};

  String[] descriptions = {"A text to describe Canadian " +
    "flag is omitted", "British flag ...", "American flag ...",
    "German flag ...", "Indian flag ...", "Chinese flag ..."};

  try {
    // Create a prepared statement to insert records
    PreparedStatement pstmt = connection.prepareStatement(
      "insert into Country values(?, ?, ?)");

    // Store all predefined records
    for (int i = 0; i < countries.length; i++) {
      pstmt.setString(1, countries[i]);

      // Store image to the table cell
      java.net.URL url =
        this.getClass().getResource(imageFilenames[i]);
      InputStream inputImage = url.openStream();
      pstmt.setBinaryStream(2, inputImage,
        (int)(inputImage.available()));

      pstmt.setString(3, descriptions[i]);
      pstmt.executeUpdate();
    }

    System.out.println("Table Country populated");
  }
  catch (Exception ex) {
    ex.printStackTrace();
  }
}
```

```java
private void retrieveFlagInfo(String name) {
  try {
    pstmt.setString(1, name);
    ResultSet rs = pstmt.executeQuery();
    if (rs.next()) {
      Blob blob = rs.getBlob(2);
      ByteArrayInputStream in = new ByteArrayInputStream
        (blob.getBytes(1, (int)blob.length()));
      Image image = new Image(in);
      ImageView imageView = new ImageView(image);
      descriptionPane.setImageView(imageView);
      descriptionPane.setTitle(name);
      String description = rs.getString(2);
      descriptionPane.setDescription(description);
    }
  }
  catch (Exception ex) {
    System.err.println(ex);
  }
}
```