JDBC Programming

- JDBC Fundamentals
- Connection Pooling
- Querying a ResultSet
- Prepared Statements

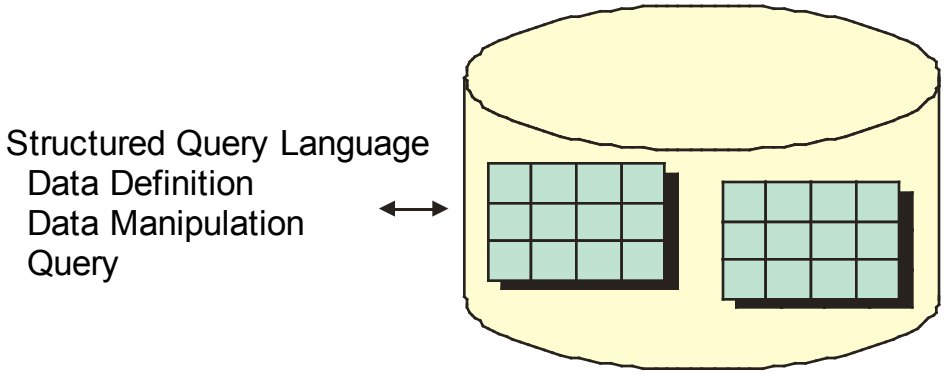
In this chapter, you will learn how servlets can use JDBC to access data stored in relational databases.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Database Fundamentals

- This chapter focuses on relational databases, which are composed of tables that can be joined together
- The standard "language" for relational database manipulation is SQL



2 - 2

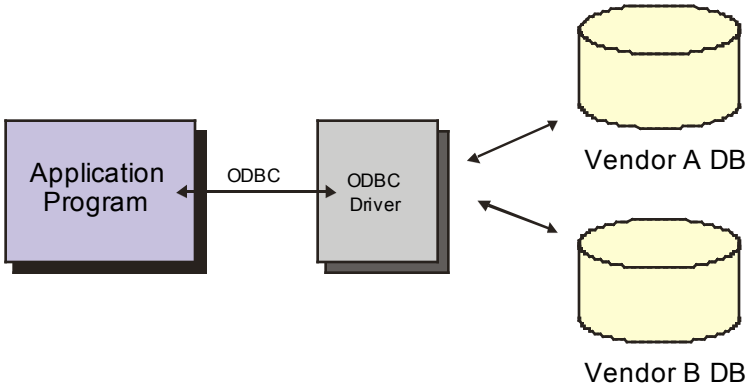
A relational database consists tables of information that you can join or relate together to query information. The concept of relational databases was invented by Edgar Codd at IBM San Jose in the early 1970s and popularized by Oracle.

Starting in the early 1990s, the ANSI and ISO committees jointly worked to define SQL as a standard language for manipulating relational data. Most modern relational databases adhere to the SQL standard.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to ODBC

- Most database vendors define a proprietary API for accessing the database
- Microsoft created the ODBC API to insulate application programs from vendor-specific API
- ODBC is a C-based API and uses pointers heavily (not suitable for Java)



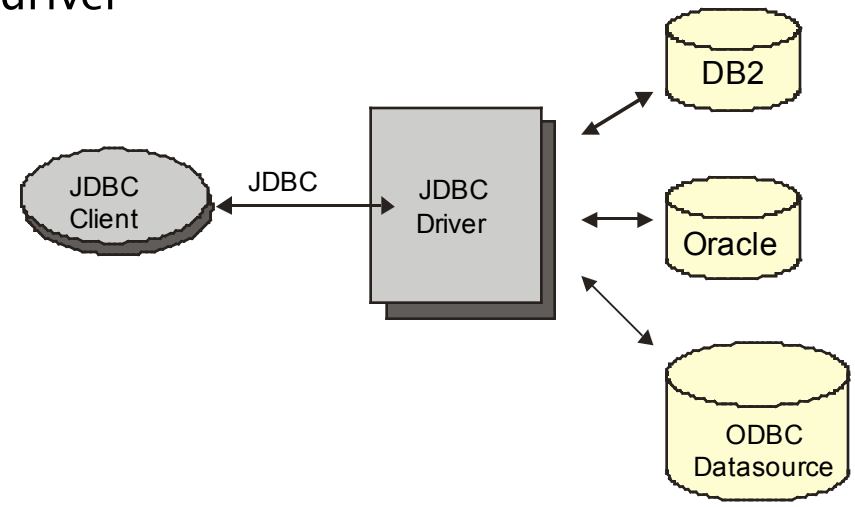
ODBC was developed by Microsoft in the early 1990s to provide a database-neutral programming interface for Windows applications. ODBC insulates applications from database specifics using ODBC drivers, which are generally written by the database vendor. In other words, the program talks to the driver, not the database -- thus the program should be portable across databases.

The problem with ODBC for Java is that ODBC was originally written for the C programming language and uses C language features that don't map well to Java.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to JDBC

- Like ODBC, JDBC lets you connect to relational databases using a vendor-independent API
- Each database vendor must provide a JDBC driver



2 - 4

JDBC is similar in concept to ODBC -- it provides a database-neutral API so that Java programs can access relation data. Like ODBC, JDBC has the notion of drivers that insulate the application from the database. In other words, your JDBC program, referred to as the JDBC client), uses the API provided by the driver.

Note that even though the program using JDBC is referred to as a "client", it doesn't necessarily have to run on a client computer -- a JDBC client is any program that uses the JDBC API. In fact, in this course, our JDBC clients will be servlets.

Note also that if you need to access a database that has an ODBC driver but no native JDBC driver, you can use the JDBC-ODBC bridge driver supplied by Sun. You should avoid the bridge if possible, since it's not known for fast performance.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What Can You Do with JDBC?

- JDBC supports the normal activities you'd expect from a database program, for example:
  - Creating records, retrieving records (queries), updating records, deleting records. These are sometimes referred to as CRUD
  - Accessing database metadata (e.g. table names, table column names)
  - Calling stored procedures
- JDBC is not really object-oriented, however

CRUD stands for Create Retrieve Update and Delete.

JDBC lets you access the raw data stored in a database, but it does not let you treat database rows directly as objects. For example, if you have a Student database table, JDBC lets you read and write the Student information from the table, but it's your job, if you desire, to define and create Student objects</i> from the rows in the table.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

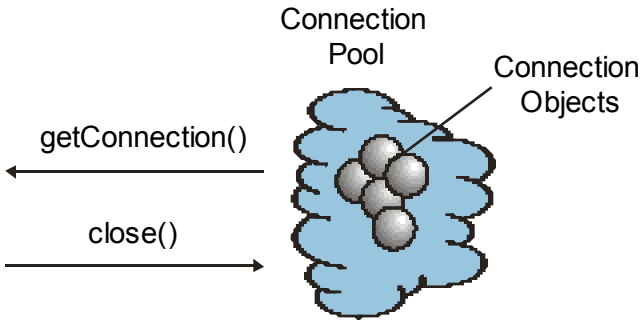
JDBC History

- The JDBC 1.0 API was originally introduced in JDK 1.1 but is now a core technology of Java2 Enterprise Edition
- JDBC 2.0 API adds support for *connection pools* (in the so-called Optional Package), performance enhancements and additional APIs
- JDBC 3.0 adds support for non-relational data sources (e.g. text files)

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Connection Pools

- To avoid expensive creating and closing connections, some JDBC drivers implement the JDBC 2.0 connection pooling feature
- This is especially helpful in Web applications since there's a lot of short-lived access to databases



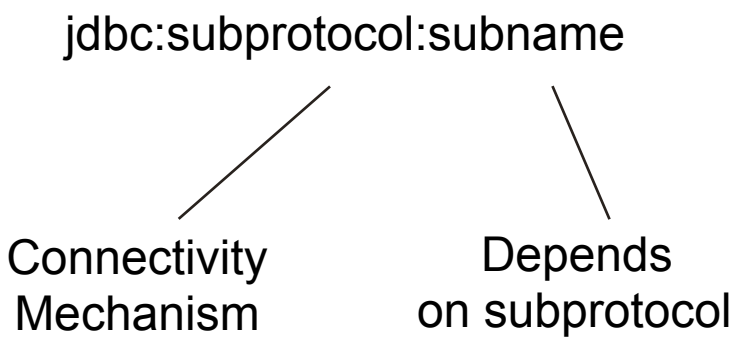
Connection pooling is almost a requirement for Web applications. To support pooling from a servlet, you need both a compliant JDBC driver and a container that supports pooling.

The way pooling works is that the container pre-creates a number of connections at startup. When a servlet needs a connection, the container retrieves a connection from a pool. When the servlet is finished using the connection, the close method does not destroy the connection, but simply returns it to the pool.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JDBC URLs

- JDBC uses URLs to represent data sources
- Much of the URL syntax is vendor-specific
- Examples:
  - jdbc:db2:MyDb
  - jdbc:hsqldb:hsql://localhost



To reference a data source, JDBC programs specify a URL, similar to the URL for a web page. The URL contains three parts: the protocol, which is "jdbc:", the sub-protocol and the sub-name. The last two parts depend on which JDBC driver you are using. The sub-protocol essentially identifies the driver, while the sub-name locates the data source itself.

This page shows two URLs for different databases. The first is for IBM's DB2 -- the sub-name "MyDb" is the name of a DB2 database. The second example is for the free HSQLDB database. Sub-names for that database reference a URL for the database server.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

SQL Exceptions

- Most JDBC operations throw the SQLException if a database error occurs
- In JDBC 2.x, the JDNI API may also throw the NamingException

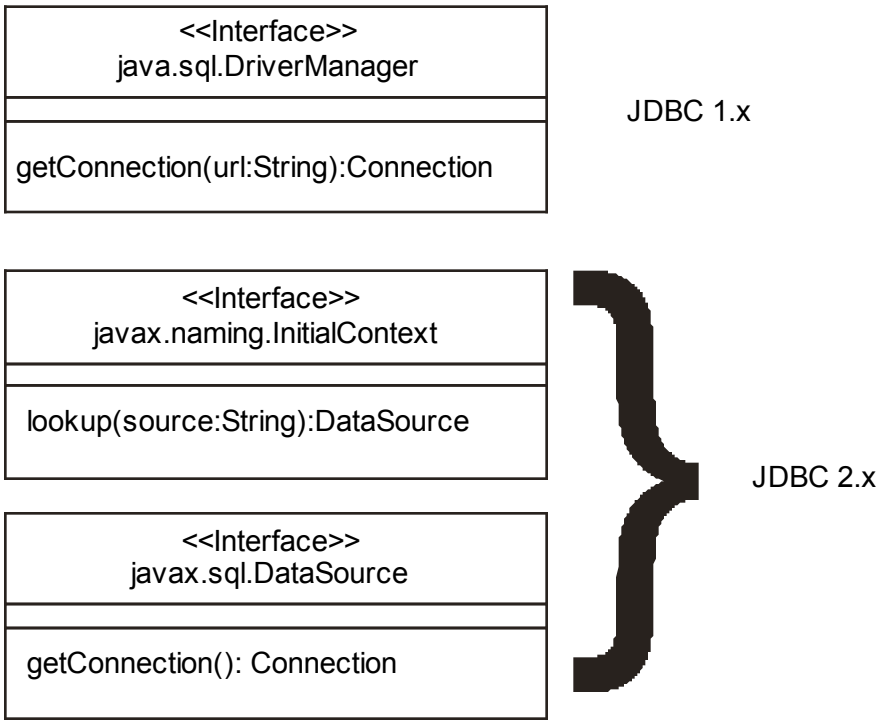
```
try
{
  // some JDBC call(s)
}
catch ( java.sql.SQLException exc )
{

}
```


Whenever you work with JDBC, your program needs to be prepared for exceptions. This page shows some sample code that executes some JDBC statements and catches the SQLException.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The JDBC API for Data Sources



There are two ways to connect to a data source in JDBC depending on whether your driver (and container) support the JDBC 2.0 Optional Package.

The JDBC 1.0 uses the DriverManager.getConnection static method to retrieve a connection. This method accepts the JDBC URL and returns an object of type Connection which you can use to access the data source.

Before using the JDBC 2.0 Optional Package API, an administrator must create a Data Source definition using tools provided by the application server vendor. The Data Source definition includes the URL for the data source. The administrator also assigns a name to the Data Source and registers the name in the Java Naming and Directory Interface (JNDI) namespace. Then, a JDBC client can look up the Data Source using JNDI and retrieve a Connection object for the data source.

Note that regardless of which technique you use, you end up with a Connection object with which you can access the relational data.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Opening a Connection, JDBC 1.x

```
1    Connection con = null;
2    try
3    {
4      Class c =
5         Class.forName ("org.hsql.jdbcDriver");
6      con =
7         DriverManager.getConnection (
8             "jdbc:HypersonicSQL:c:\\test" );
9    }
10   catch ( ClassNotFoundException exc )
11   {
12     System.out.println (
13         "Unable to load driver: "  + exc );
14   }
15   catch (SQLException exc)
16   {
17     System.out.println (
18        "Unable to connect: "  + exc );
19   }
```

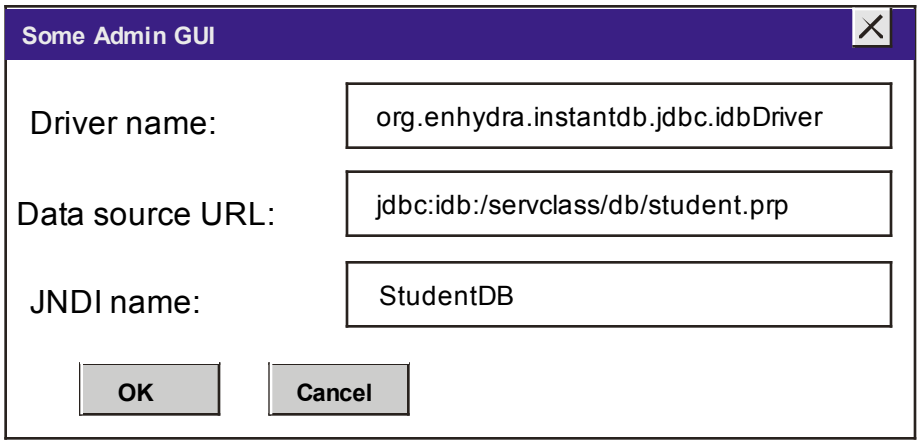
Once you are assured that the JDBC driver's available, then you can call the static "getConnection" method on the DriverManager type, which is part of the JDBC API. You pass this method a URL which names the data source.

If something goes wrong (e.g. the URL syntax is incorrect), the JDBC driver will throw an exception.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring a Datasource, JDBC 2.x

- In JDBC 2.x, an "administrator" must configure each datasource using tools provided by the application server vendor
- The configuration specifies the datasource's Java Naming and Directory Interface (JNDI) name and JDBC URL



Now let's switch gears and talk about using JDBC 2.0 Optional Package. This figure shows a stylized user interface for configuring a Data Source and registering it into the JNDI namespace. Note that we specify the JDBC driver name, here for InstantDB, specify the JDBC URL and assign the JNDI name. The JDBC client will use "StudentDB" to retrieve the Data Source.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Locating a Datasource, JDBC 2.x

- Before you can open a connection in JDBC 2.x, you must first look up the datasource using the JNDI name

```
1    DataSource ds = null;
2    try
3    {
4        InitialContext ctx = new InitialContext();
5        ds = (DataSource)ctx.lookup (
6                "java:comp/env/jdbc/StudentDB" );
7    }
8    catch ( NamingException exc )
9    {
10       System.out.println ( "Bad lookup" +
11                            exc );
12   }
```

This code shows how to use JNDI to retrieve the DataSource defined on the last page. In line 4, we instantiated an InitialContext object, which acts as a gateway to JNDI. In lines 5 and 6, we then use the lookup method to retrieve the DataSource, using the name assigned by the administrator.

Note that the name string contains more than just the text "StudentDB" -- it is prefixed by the "java:comp/env/jdbc/" context string. This prefix string is somewhat dependent on your application server, so consult its documentation for details. The prefix string shown here works for the JRun application server.

The InitialContext type is defined in the javax.naming package.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Opening a Connection, JDBC 2.x

- Once you have retrieved a datasource reference, you can open a connection
- If the datasource implements connection pooling, this may not actually create a new connection

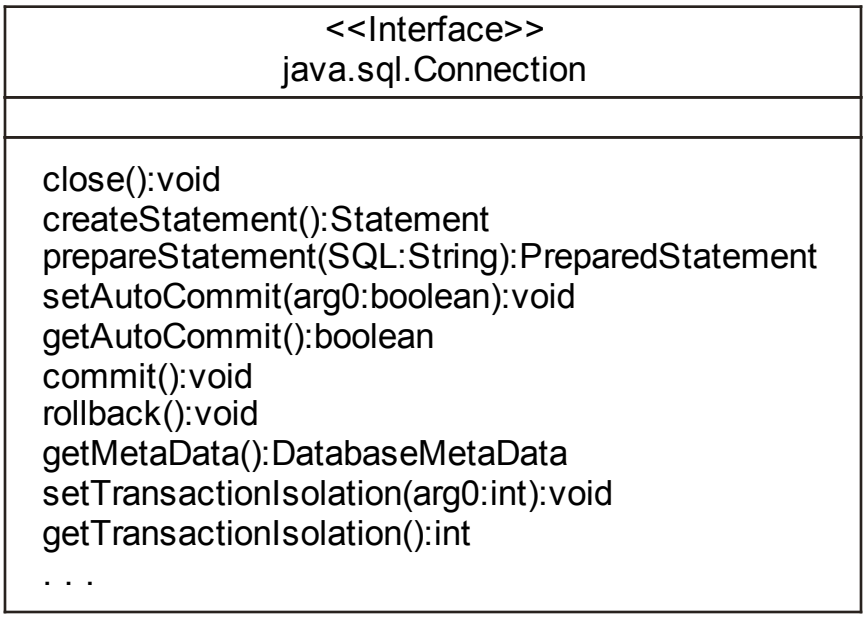
```
1    Connection con = null;
2    try
3    {
4        con = ds.getConnection();
5    }
6    catch ( SQLException exc )
7    {
8        System.out.println ( "Unable to connect"  +
9                             exc );
10   }
```

After retrieving a DataSource, a JDBC 2.0 client can retrieve a Connection object using the DataSource getConnection method as shown in line 4.

Note that there are other overloaded versions of getConnection with additional parameters, such as for database username and password.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The JDBC API for Connections



Whether you are using JDBC 1.0 or JDBC 2.0's Optional Package, you end up with a Connection object. This diagram shows some of the more useful methods from the Connection interface.

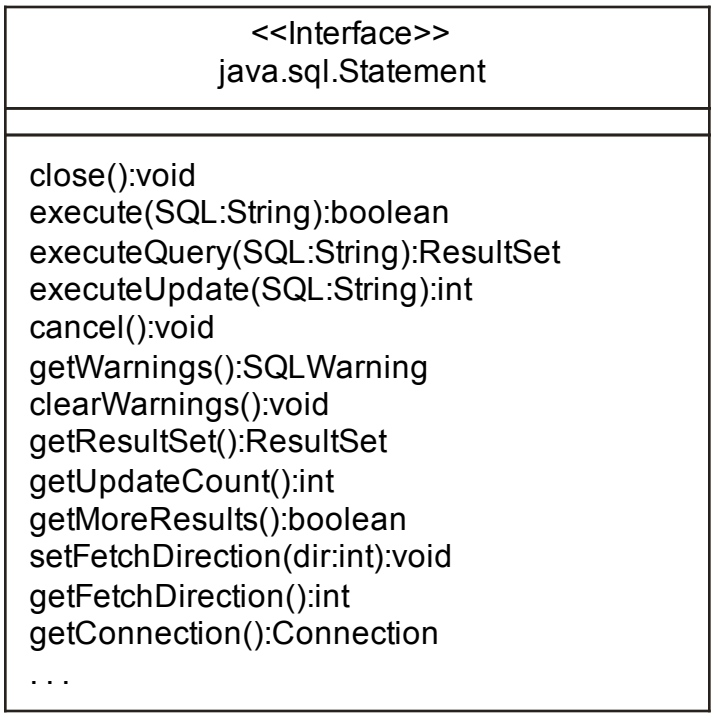
You should be sure to close a Connection when you are finished with it to prevent memory leaks. Recall that if you are using connection pooling, the close method returns the object to the pool instead of destroying it.

One particularly useful method is createStatement, which returns an object of type Statement. As you will see on the next few pages, you can use the Statement object to perform database operations, such as queries and updates. The PreparedStatement type is similar to Statement, but may have performance benefits. We will cover it later.

Also note that Conneciton type defines several methods that work with transactions. We will cover transactions in more detail later.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The JDBC API for Statements



The Statement type provides several methods for manipulating a database. For example, executeQuery accepts an SQL query string and returns a ResultSet object that represents the rows matching the query. We will see more about it in a moment.

Another useful method is executeUpdate, which accepts an SQL string specifying a command to modify the database, for example to insert a new row. The executeUpdate method returns an integer that contains the count of rows affected.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Executing a Simple Query

- To execute a query, you can create a Statement and issue the executeQuery method, passing the SQL "select" statement

Student

studentID	name	GPA
12	Bill	3.45
45	Sue	3.22

```
1    Statement stmt = null;
2    ResultSet rs = null;
3
4    try
5    {
6        stmt = con.createStatement();
7        rs = stmt.executeQuery ( "SELECT * FROM student" );
8    }
9    catch ( SQLException exc )
10   {
11       System.out.println ( "Query error: "
12                           + exc );
13   }
```

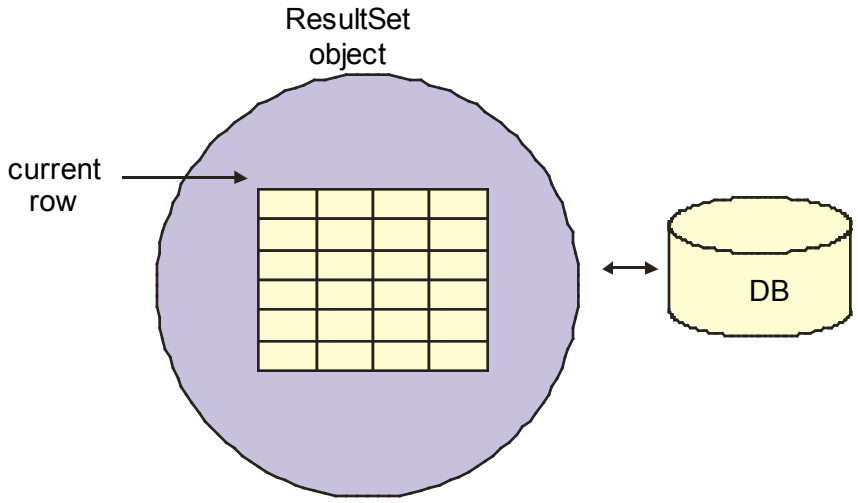

Now let's see some JDBC code to process a simple query. Assume that we've opened a data source containing a table named Student as shown in the figure. The Student table contains three columns of various types.

To query all rows, all columns from the Student table, in line 6, the program creates a Statement from the Connection, and then in line 7 calls executeQuery, passing the SQL string to query all rows, all columns. This method returns a ResultSet object -- we will see how to process it on the next few pages

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Processing a ResultSet

- The ResultSet acts as a collection of the matching rows -- you can iterate through the results one row at a time

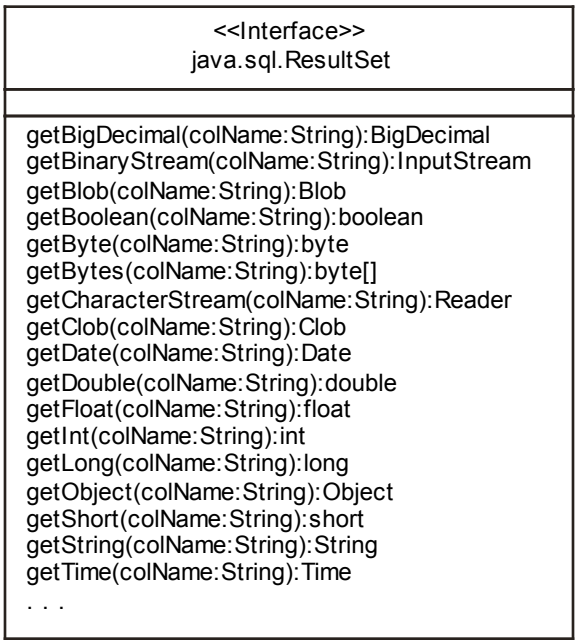


The ResultSet object represents the returned rows from a query. It contains the notion of a "current row" -- when the ResultSet is first created, the current-row pointer points to "above" the first row. To retrieve data, you loop and each time through the loop, you move the current-row pointer to point to the next row. To process a row, you can use APIs to retrieve the data for a given column on the current row.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

SQL Types and Java Types

- Since Java types and SQL types are not identical, JDBC must convert
- You can call getXXX() methods on a ResultSet row and column to force a conversion



Unfortunately, the data types used by Java and by databases are not identical. Therefore, to use data from a database, you will need to convert the data into a Java type. To help, the ResultSet type provides a number of "get" methods that accept a column name or number and convert the data to a particular type. For example, getString ("studentID") would return the studentID column from the current row as a Java String. This would work even though the studentID was stored as some numeric type in the database, for example INTEGER. If the conversion is inappropriate, the "get" method will throw an exception.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Processing a ResultSet, cont'd

- You can retrieve a column's data either by using the column's name or column index (starting at zero)

```
1    try
2    {
3        while ( rs.next() )
4        {
5          int studentID = rs.getInt ( "studentID" );
6          String studentName = rs.getString ( "name" );
7          float gpa = rs.getFloat ( "GPA" );
8        . . .
9        }
10   }
11   catch ( SQLException exc )
12   {
13       System.out.println ( "Error: " + exc );
14   }
```


Now let's see some code that processes the ResultSet for a query from the Student table shown earlier. In line 3, we start a loop that iterates the ResultSet. We stay in the loop until the next method hits the end of the ResultSet, at which point it returns false.

In lines 5 to 7, we retrieve columnar data from the current row, converting to the Java types of int, String and float. The ellipsis in line 8 represents doing something with the data -- if this code was in a servlet, perhaps we could generate an HTML table row here.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Closing a Connection

- When you are finished with a connection, you should close it to free resources
- If you are using connection pools, calling "close" may not actually close the connection

```
1    Connection con = null;
2    try
3    {
4       con = . . .;
5       . .
6    }
7    catch (SQLException exc)
8    {
9       // report error
10   }
11   finally
12   {
13      if (con != null) con.close();
14   }
```


To avoid tying up the database, you should always close the Statement and Connection when you are finished with them. Note that you don't explicitly need to close a ResultSet -- closing the Statement does that automatically. Also note that if you do another execute method on a Statement, that automatically closes the ResultSet, too.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Inserting a Row

- To insert a new row, create a Statement and issue executeUpdate with the SQL "insert" command as the argument
- The executeUpdate method returns the number of rows inserted

```
1    try
2    {
3        String cmd =
4            "INSERT INTO student VALUES (" +
5            "12, 'Paul', 3.54)";
6
7        Statement stmt = con.createStatement();
8        int rows = stmt.executeUpdate ( cmd );
9    }
10   . . .
```


JDBC is not limited to reading a table. Here we demonstrate using the executeUpdate method to insert a new row into the Student table. Note that constructing the SQL command string is tedious and error-prone due to the fact that you must "escape" quotes for SQL character data. We recommend that you construct the string in a separate String as we did here rather than embedding it directly in the executeUpdate method -- that way, if you're having problems, you can easily write the String to a log so you can debug it.

The executeUpdate command returns the count of rows updated. In this example, it should return the number one to indicate that a single row was inserted.

For simplicity, here we hard-code the values (e.g. 3.54) but in a real program they would probably be in variables. In addition, if this was from a servlet, the data would likely come from an HTML input form.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Updating a Row

- To modify existing row(s), create a Statement and issue executeUpdate with the SQL "update" command as the argument
- The executeUpdate method returns the number of rows updated

```
1    try
2    {
3        String cmd =
4            "UPDATE student SET gpa = 3.87" +
5            " WHERE studentID=12";
6
7        Statement stmt = con.createStatement ();
8        int rows = stmt.executeUpdate ( cmd );
9    }
10   . . .
```

If you have an existing row in a table that you wish to change, you can use JDBC to issue an Update command. In lines 3 to 5, this code defines a String that modifies the "gpa" column of a particular row in the Student table. The "WHERE" clause selects the Student in question.

As with the Insert command shown on the last page, the executeUpdate method returns the number of rows updated, which should be one assuming that "studentID" is the unique primary key of the Student table.

And as on the last page, here we hard-code the values, but a real program would use variables.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Removing a Row

- To get rid of row(s), create a Statement and issue executeUpdate with the SQL "delete" command as the argument
- The executeUpdate method returns the number of rows deleted

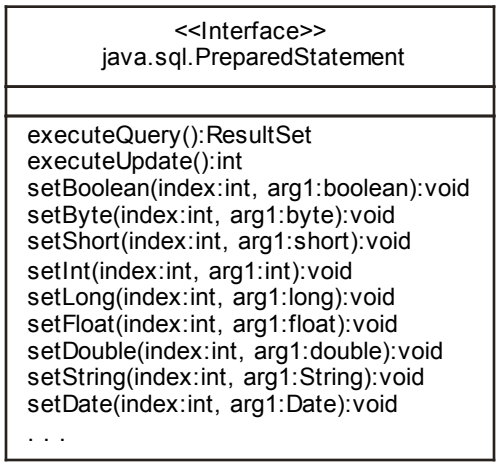
```
1    try
2    {
3        String cmd =
4            "DELETE FROM student" +
5            " WHERE studentID=12";
6
7        Statement stmt = con.createStatement ();
8        int rows = stmt.executeUpdate ( cmd );
9    }
10   . . .
```


This code shows hot to remove a row from a table. Again we hard-code the "WHERE" clause to select the Student in question.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Prepared Statements

- Using prepared statements can increase performance, since they are precompiled
- You typically precompile the statement and then use it multiple times, specifying different parameters each time
- Prepared statements are also less susceptible to **SQL injection** attacks



PreparedStatements are especially useful if you plan to use the same sort of SQL command multiple times. With a PreparedStatement, you write the basic SQL command with placeholders for arguments that you can substitute before you execute the command.

Like most things in JDBC, the performance benefit of PreparedStatements is quite dependent on the database in question and its JDBC driver. But potentially, your database operations can run more efficiently if you use PreparedStatements instead of simple Statements. You should benchmark with your own access patterns to choose which approach provides the best performance for your application.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Prepared Statements, cont'd

```
1    int ID = 12; double gpa = 3.87; String name = "Lynn";
2
3    PreparedStatement ps = null;
4    try
5    {
6        String cmd =
7            "UPDATE student SET GPA=?, " +
8            "NAME=? WHERE studentID = ?";
9
10       ps = con.prepareStatement ( cmd );
11       ps.setDouble ( 1, gpa );
12       ps.setString ( 2, name );
13       ps.setInt ( 3, ID );
14
15       int rows = ps.executeUpdate();
16   }
17   . . .
```


This code demonstrates a PreparedStatement. In line 3, we define a reference to a PreparedStatement -- we initialize the reference in line 10, using the String from lines 6 to 8. In that String, we write an SQL "update" command, using the question mark character to indicate a replaceable parameter. Before we actually execute the command, in lines 11 to 12, we call methods on the PreparedStatement to provide the values for the replacement parameters. Note that parameter indexes start at one, not zero!

In a real program, we would not hard-code the parameter values as we did here. If this code was in a servlet, perhaps we would read the parameter values from an HTML form.

Note one other benefit of PreparedStatements as compared to simple Statements -- we don't have to worry about "escaping" quoted character data -- see line 12, which substitutes a String. That makes the code a bit easier to read and maintain.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About the architecture of JDBC
- How to use the JDBC API to access relational data sources

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum