



CHAPTER 2

JDBC FUNDAMENTALS



OBJECTIVES

After completing “JDBC Fundamentals,” you will be able to:

- Know the main classes in the JDBC API, including packages **java.sql** and **javax.sql**.
- Understand the differences between SQL data types and native Java data types.
- Know the difference between JDBC driver types and choose the one appropriate for a project.
- Use **DriverManager** to create a database **Connection**.
- Perform a simple query using the **Statement** interface.
- Retrieve data from a **ResultSet**.
- Understand and handle SQL NULLs.
- Properly handle database resources, warnings, and exceptions.
- Understand the differences in SQL data types, Java data types and how to perform conversion between them.
- Use a JDBC wrapper class to handle opening and closing database resources and SQL exception handling.

What is the JDBC API?

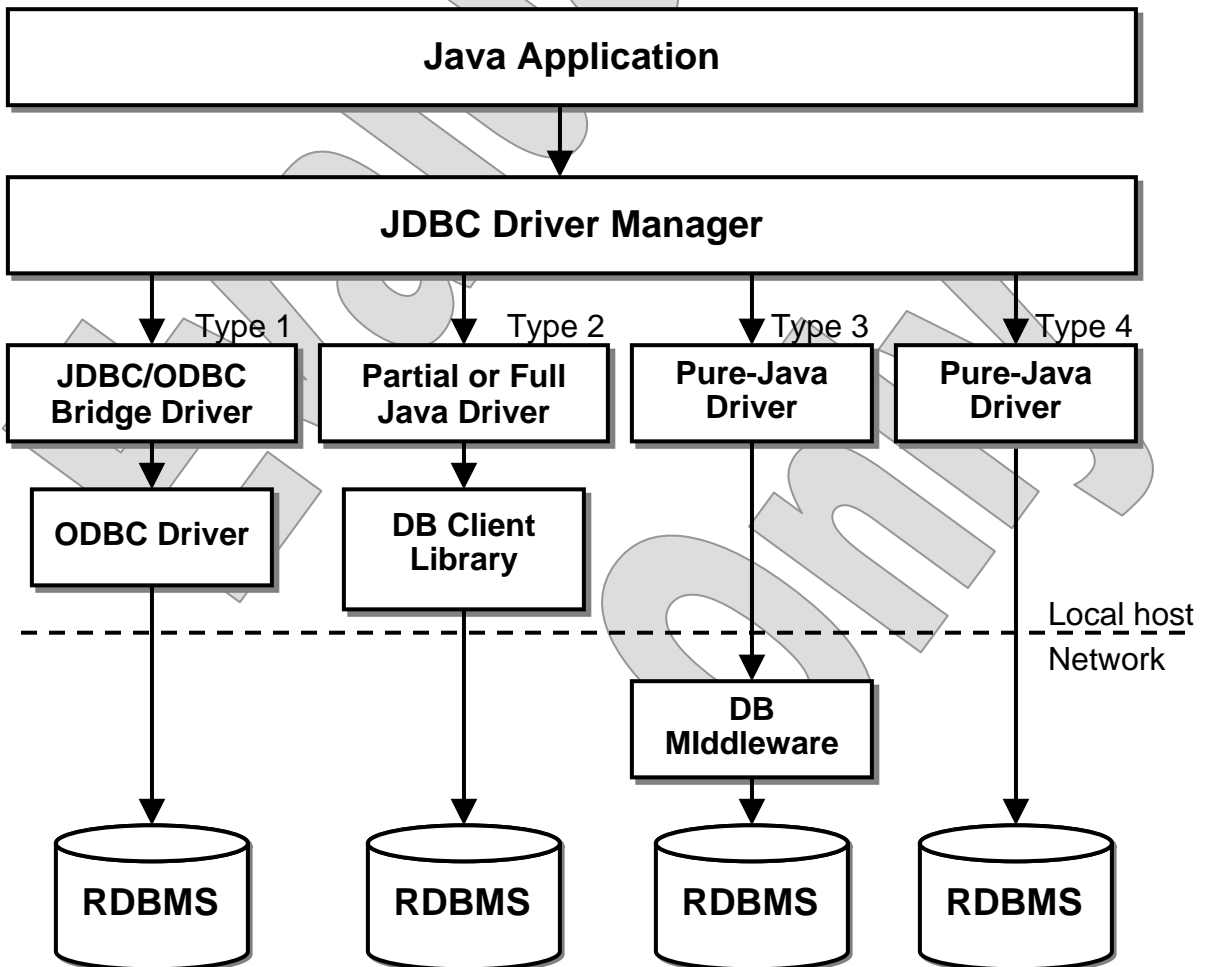
- **Java Database Connectivity (JDBC™) allows for accessing any form of tabular data from any source.**
 - Relational databases are most common, but JDBC drivers for XML, Excel, or legacy data sources can be obtained.
 - There is a set of classes and interfaces for database or tool developers bundled with the J2SE Core API.
 - It allows for easy information dissemination.
 - Object persistence can be built on top of JDBC.
 - JDBC is built as an object oriented Java alternative to ODBC, but is easier to learn and is more powerful at the same time.
 - Drivers must support ANSI SQL 92 Entry Level at the minimum. Most drivers also have some support for SQL 99.
- **The `java.sql` package contains the core JDBC API.**
 - It includes basic support for **DriverManager**, **Connection**, **Statement** and its children, and **ResultSet**.
 - Metadata support for the database and result sets are supported for advanced use.
- **The `javax.sql` package contains the JDBC Optional Package.**
 - It was added to JDBC 2.0, but incorporated in JDBC 3.0.
 - JDBC includes support for data sources, row sets, and XA support.

Basis for Other APIs

- **SQLJ (originally JSQL) ANSI Standard is supported by Sun, IBM and Oracle.**
 - Oracle implemented SQLJ using a cross-compiler that converted it to JDBC calls.
 - Oracle dropped support for SQLJ in 2004 and it is not widely used today.
 - While IBM and Sun gave tacit support, only IBM implemented it in its products.
- **Enterprise Java Beans (EJB) offer object persistence through entity beans.**
 - This persistence is usually implemented using JDBC at the application server level.
 - The implementation is invisible to the developer.
 - EJB Query Language is patterned after SQL and might use JDBC in the implementation layer.
- **JDO (Java Data Objects), JSR 12, is becoming popular as an alternative to using EJBs for persistence.**
 - JDO allows for object persistence without directly using a lower level API like JDBC.
 - Some, but not all, implementations of JDO use JDBC.

JDBC Framework

- The **JDBC DriverManager** is implemented in the JDBC API included in the JDK.
- The drivers are written by the database vendors, but implement interfaces in the **java.sql** and **javax.sql**.
 - In general, each driver is written for a specific database and a specific database may have more than one type of driver.
 - The JDBC/ODBC Bridge driver was written by Sun to interface to existing ODBC drivers at a time when no native JDBC drivers existed.

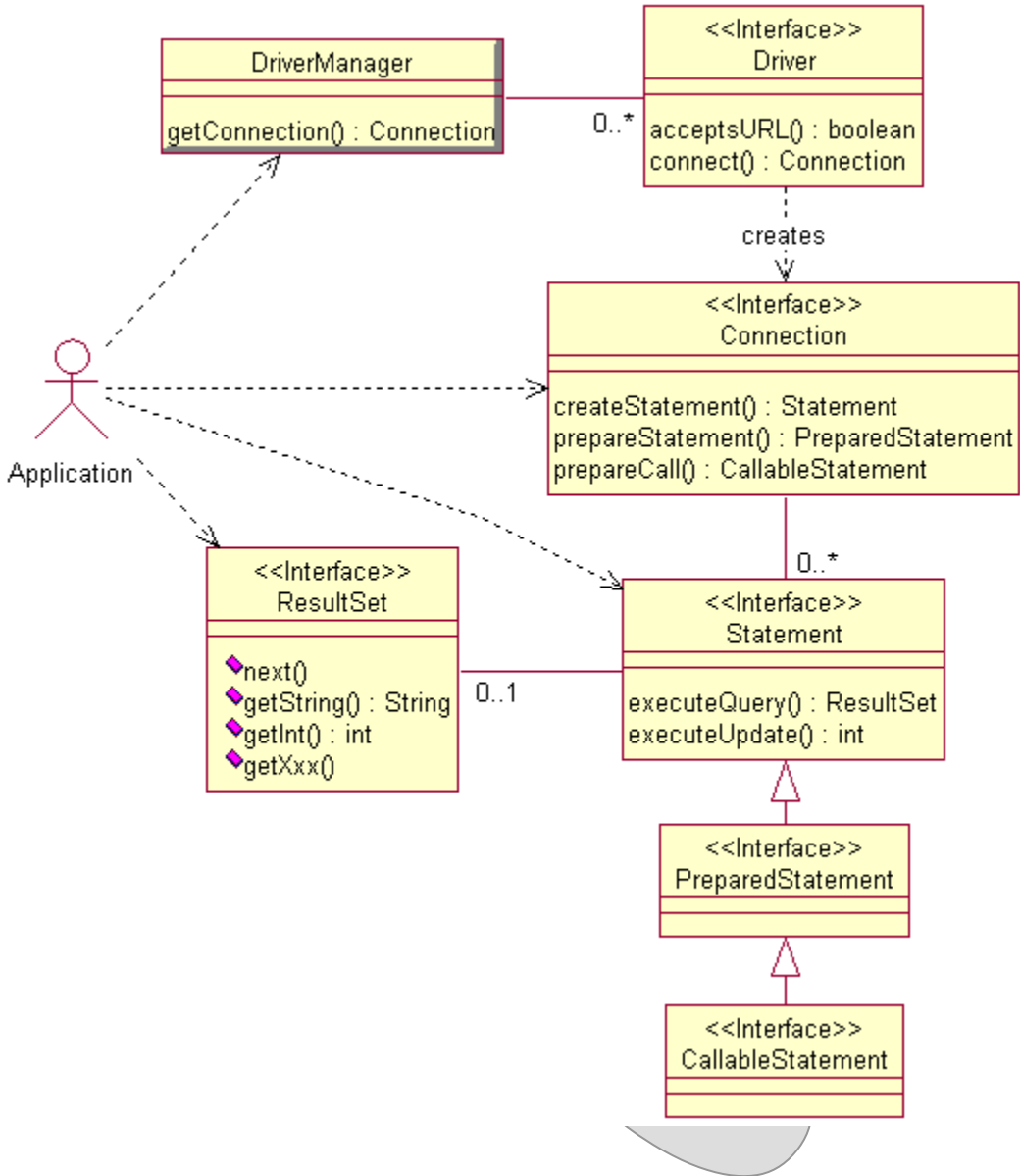


JDBC Drivers

- The database vendor implements **JDBC Driver**.
- **DriverManager** is implemented in the core API as a static class to load database-specific drivers.
- Driver classes are typically implemented by database vendors and packaged as JAR files.
 - A JDBC/ODBC bridge is the only driver included with the Java SDK.
 - All other drivers have to be acquired from the database vendor, or in some cases a third party.
- **Connection** represents a connection to the database.
- **Statement** and its subclasses hold the SQL statement string that will be sent to the database.
 - **PreparedStatement** and **CallableStatement** will be examined in the next chapter.
 - All methods for executing statements will close the current **ResultSet** object(s) before returning new ones.
- **ResultSet** objects represent the results returned from the database.

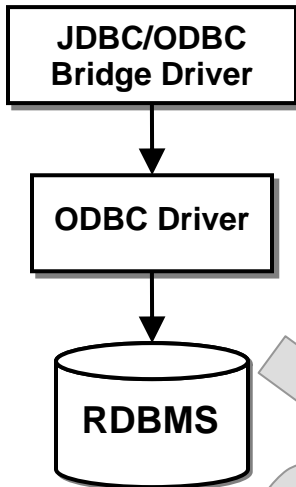
JDBC Interfaces

- The following UML diagram summarizes the JDBC driver interfaces most useful to application code:



JDBC Driver Types: Type 1

- There are four driver types defined in the JDBC specifications, known simply as types 1 through 4.
- Type 1 is a JDBC-to-native-API bridge.

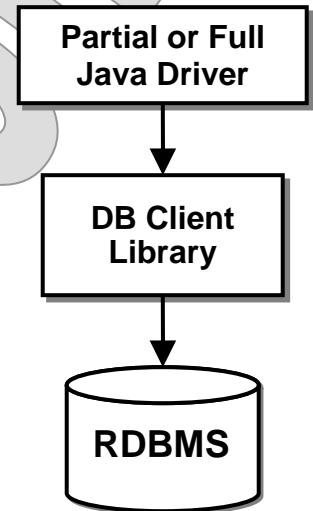


- This driver is more commonly known as the **JDBC-ODBC Bridge**, though technology other than ODBC could be used.
- The actual JDBC-ODBC bridge, implemented in **java.sql** and delivered with the JRE, allows JDBC access via ODBC drivers – which in turn must be installed on the client.
- It is useful in situations where a native JDBC driver does not exist for the database.

- Since this is a **two-step process** to communicate with a database, it is never as efficient as the other driver types.

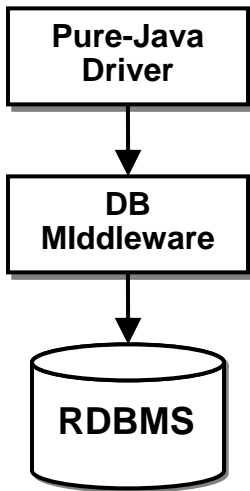
JDBC Driver Types: Type 2

- Type 2 is a driver that converts JDBC calls to calls on a local database-client code library.
 - It may be implemented partly in Java and partly in native code, or completely in Java.
 - Oracle’s “thick” driver is an example. It uses the Oracle Call Interface (OCI) to communicate with the database.
 - It is commonly used on middleware servers where driver installation and update are not a problem.
 - This may be the fastest driver available for production use, but testing with other driver types is always a good idea.



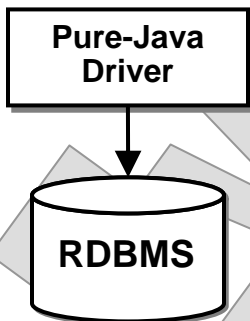
JDBC Driver Types: Types 3 and 4

- Type 3 is a pure-Java network driver.



- It translates JDBC calls into RDBMS-independent calls to a **middleware server**, which in turn communicates with the database.
- While this allows for flexibility, it is a two-step process, and therefore less efficient.
- It is often used to communicate with legacy data stores.

- Type 4 is a native-protocol, pure-Java driver.



- It converts JDBC calls directly into the **native protocol** of the target database.
- This allows for **direct communication** between the client and database server.
- It is the most popular driver for major databases.
- Oracle's 'thin' driver is an example.
- It can be faster than the 'thick' Oracle driver depending on the nature of the application.

Obtaining JDBC Drivers

- Sun has a central list of drivers for various databases:

<http://servlet.java.sun.com/products/jdbc/drivers>

- This list is somewhat outdated.

- MySQL drivers are found here:

<http://www.mysql.com/products/connector/j/>

- Oracle drivers are found here:

http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/

- The Apache Derby database/driver combination is found here:

<http://incubator.apache.org/derby/index.html>

- Derby includes the driver in the same jar file with database engine. Derby is based on IBM Cloudscape, version 10.0. Derby is in ‘incubator’ status at Apache as of this writing, and will be moved to a new URL at some point.

- PostgreSQL drivers are found here:

<http://jdbc.postgresql.org/download.html>

- Generally you should use the latest drivers available,

- Almost without exception, they are backwards compatible with older versions of the database. While new drivers may solve many compatibility problems and bugs, testing with new drivers is always a sound practice.

Loading Drivers

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
```

- JDBC requires that a driver class register itself with **DriverManager**, when the driver class is loaded.
- The usage above triggers the loading of the driver class, without creating a driver instance directly.
- The driver documentation will identify the string value to use for the driver class.
- **Class.forName** throws only one exception: **ClassNotFoundException**.
- A couple of **Connection** problems might be:
 - “Class not found” – This message usually means the driver is not located on the CLASSPATH.
 - “Driver not found” – This rarely happens any more unless you are using a legacy driver. If so, then you will need to code the line as `Class.forName(driver).newInstance`.
- There are better practices than the above, most aimed at removing the driver class name from application code.
 - Derive the class name from a command-line argument or system property.
 - Use a **DataSource** instead of **DriverManager**; this technique will be beyond our scope for this course.

Making the Connection

```
Connection conn = DriverManager.getConnection  
("jdbc:derby:/MyDatabase", "me", "mypassword");
```

- The line above is the most commonly used form for the **getConnection** method.
 - If the target database does not need a username and password, there is an overload of this method that takes only the URL string.
- Notice you are creating the **Connection** object with the help of the **DriverManager** object rather than instantiating it with **new**.
 - This is the **factory pattern**, by which one object or utility provides instances of another class.
 - You will see this pattern repeated through the API.

JDBC URLs

- While the **getConnection** step is very easy, it is the most error prone part of JDBC.
- Getting the URL, user name and password right can be a challenge.
- A JDBC URL is coded as
`jdbc:subprotocol:subname`
 - The RDBMS vendor will typically be represented in the **subprotocol**, as in “jdbc:derby:”, “jdbc:mysql:”, “jdbc:oracle”, “jdbc:postgresql”.
 - From there, every vendor has it’s own interpretation of what the rest of the URL means!
 - Again, get this value from the driver documentation.
- Getting the URL wrong will produce errors such as “no suitable driver”, “invalid URL specified”, “connection error”, “IO exception”, “unknown error”, “listener refused the connection” and several others.
 - Derby might say “Database not found” if the path is wrong.
- You will also get errors at this point if the RDBMS or database is not started or is otherwise inaccessible.
 - They will be something similar to “connection refused”, “network adapter could not establish the connection”, or “user not found”.

Making the Connection

- The second parameter to **getConnection** is the user name.
 - On some operating systems, the database will authenticate the user based on the OS user.
 - Getting this value wrong will produce errors such as “invalid authorization specification” or “username and/or password are invalid”.
- The third parameter is the given user’s password.
 - On some operating systems, the database will authenticate the password based on the OS password.
 - Invalid password will give the same errors as user name above plus “access is denied”.
- URL strings for the four supported databases in this course are:

`jdbc:derby:/Capstone/JDBC/Database/earthlings`

`jdbc:mysql://localhost/earthlings`

`jdbc:oracle:thin:@localhost:1521:orcl`

`jdbc:postgresql:earthlings`

In this lab you will edit database driver strings, test the database connection, and set these strings as user properties for future labs.

Detailed instructions are contained in the Lab 2A write-up at the end of the chapter.

Suggested time: 10 minutes for Derby or 30 minutes for MySQL, Oracle and PostgreSQL.

Evaluation Only

Creating the Statement

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

- Creating the **Statement** object is simple and uneventful.
- The **Statement** is created from the **Connection** object using the **createStatement** method.
- It is used to send a SQL query to the database.
- You do not supply a database query string at this point. That will be done when you execute it.
- A connection can create (and support) any number of **Statement** instances.

Executing the Statement

```
ResultSet rs =  
    stmt.executeQuery("select * from locations");
```

- The **ResultSet** object is created to receive the results from the **Statement's executeQuery** method.
- The **executeQuery** method is used with **SELECT** statements and is the form most often seen in a typical application.
- The query string can be a **String** literal as above, but more typically it will be defined as a `private static String` near the top of your code (or block) for easy maintenance.
- Query strings do not end with a statement terminator, such as a semicolon.
 - The driver will supply it for you, using the terminator expected by the RDBMS.
- The query is case-insensitive, except for quoted material.
- A **Statement** supports at most one **ResultSet**.
 - You can call **executeQuery** multiple times on a **Statement**, but it will close any open **ResultSet** when you do.
- The **Statement** query string will be compiled by the database each time it is executed.

Retrieving Values from Result Sets

```
while (rs.next())
{
    String city = rs.getString("city");
    System.out.println(city);
}
```

- The **ResultSet** object receives the results from the SQL query.
- When the **ResultSet** is initially created, the cursor is conceptually pointing above the first row.
- Thus when the **next** method is called for the first time, it moves the cursor to the first row.
- When the cursor goes beyond the last row, the **next** method returns false and the **while** loop is terminated.
- You use “getter” methods (for example, **getInt** or **getString**) to retrieve the value of each column.
- The parameter to the **getXxx** method can be either a column index or column name.
 - Column indices are 1-based and relative to the selected columns in the result set, not to the original table.
 - Use indices when you don’t know the name of the column, as when processing a “SELECT *” query.
 - Column names are preferred for the sake of readability.

Now All at Once

```
private static final String JDBC_DRIVER =
    ("org.apache.derby.jdbc.EmbeddedDriver");
private static final String DATABASE_URL =
    ("jdbc:derby:/Capstone/JDBC/Database/earthlings
");
private static final String DATABASE_USERNAME =
    ("earthlings");
private static final String DATABASE_PASSWORD =
    ("earthlings");
private static String query =
    "select * from locations";
private static Connection con;
private static Statement stmt;
private static ResultSet rs;

// Code missing here for clarity

// Load the JDBC driver
Class.forName(JDBC_DRIVER);

// Connect to the database
con = DriverManager.getConnection(DATABASE_URL,
    DATABASE_USERNAME, DATABASE_PASSWORD);

// Create the Statement object
stmt = con.createStatement();

// Create the ResultSet object, executing the query
rs = stmt.executeQuery(query);

// Print all City/State pair for each location
while (rs.next())
{
    String city = rs.getString("city");
    String state = rs.getString("state");
    System.out.println(city + ", " + state);
}
```

Naming Conventions

- It is very common for JDBC programmers to have naming conventions for the JDBC objects they create.
- This makes it easier to write and maintain code.

Object	Conventional Variable Name
CallableStatement	cstmt
Connection	con, conn
DataSource	ds
DataTruncation	dt
ParameterMetaData	paramInfo
PooledConnection	pcon, pconn
PreparedStatement	pstmt
ResultSet	rs, rset
ResultSetMetaData	rsmd
RowSet	rowset, rs
RowSetMetaData	rsmd
SavePoint	save
SQLWarning	warn, w
Statement	stmt
XAConnection	xacon, xaconn
XADataSource	xads
XAResource	resource

- In their documentation, Sun has chosen to use **con** and **pcon**, while Oracle uses **conn** and **pconn**. **RowSet** will often have a lead-in character for its type (i.e. **CachedRowSet** **crs**).
- If more than one object of any one type exists in the same class, one convention is to follow it by a sequential number. Another convention is to give it a more descriptive name followed by the name in the chart.

In this lab you will create and execute a **Statement** object using a query on the **EMPLOYEES** table and display first and last name for each employee whose salary is below \$20,000 using the **ResultSet** object.

Detailed instructions are contained in the Lab 2B write-up at the end of the chapter.

Suggested time: 30 minutes.

Evaluation Only

SQL and Java Data Types

- The JDBC API defines many SQL types in **java.sql.Types**.
- Some common JDBC types mapped to the nearest Java type are shown below:

JDBC Type	Java Type
BIGINT	long
BINARY	byte[]
BLOB	Blob
BOOLEAN	boolean
CHAR	String
CLOB	Clob
DATE	java.sql.Date
DECIMAL	java.math.BigDecimal
DOUBLE	double
FLOAT	double
INTEGER	int
NUMERIC	java.math.BigDecimal
REAL	float
SMALLINT	short
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
VARCHAR	String

- See Appendix C for a complete list.
- Developers will typically map these Java types to the SQL type of the same name.
- The JDBC 3.0 specification lists the mapping of Java types to JDBC types.

Data Type Conversion

- When using the **getXxx** methods, the developer has a lot of freedom in conversion:
 - The **getXxx** methods perform type conversation as necessary, but you should check the documentation to see how this works.
 - SQL numeric types could, for example, be retrieved into a Java **String** if it is more convenient for processing.
 - Use common sense when assigning values to make sure the return type is large enough or you may get a **DataTruncation** warning.
- **SetXxx** methods generally throw exceptions when truncation occurs.
 - Even when this occurs, the data has still been sent to the database.
 - We will discuss this further when we get to transactions.

SQL NULL vs. Java *null*

- SQL NULL and Java *null* are very different animals.
- A SQL NULL is mapped to the same type as the column that it is stored in and represents no data in that field.
 - If you add NULL + 1, the answer is NULL!
- A Java **null** represents an object reference that points to nothing, or an un-initialized reference.
 - Java only supports **null** for object references, not for primitive types.
 - Thus for object types SQL NULL can be mapped to a Java **null**, but for primitive types it must be converted to something in the legal value set.
 - This means zero for numeric types, and **false** for booleans.
- It is always best to check for SQL NULL with a call to the **java.sql.ResultSet.wasNull** method if the given column allows SQL NULLs to be stored.
 - This returns true if the last column read had a value of SQL NULL.
 - A call to **getInt** method will return 0 for a NULL value.
 - By contrast, **getBigDecimal** will return a Java **null**.

SQLException

```
catch (SQLException se)
{
    while (se != null)
    {
        System.err.println(se.getSQLState());
        System.err.println(se.getErrorCode());
        System.err.println(se.getMessage());
        se = se.getNextException();
    }
}
```

- **SQLException** has four methods that you will use on a regular basis:
 - **getErrorCode** returns a vendor specific error code. (This is where, for example, you can get the Oracle ORA error number.)
 - **getSQLState** returns the **SQLState** for this exception. X/Open and SQL99 define **SQLState** values. Be aware that Oracle and some other databases could return **null** here.
 - Any **SQLException** can be chained to additional exceptions. Get the additional **SQLException** objects by calling **getNextException**.
 - **getMessage** returns the vendor-specific error message.

SQLWarning

```
SQLWarning warn = stmt.getWarnings();
while (warn != null)
{
    System.err.println(warn.getSQLState());
    System.err.println(warn.getErrorCode());
    System.err.println(warn.getMessage());
    warn = warn.getNextWarning();
}
```

- **SQLWarning** extends **SQLException**, but it is not thrown by code that generates it.
- Rather it is silently chained to the object whose method caused it.
 - A **SQLWarning** could happen to a method of any JDBC object, and there could be several of them.
 - Use **getNextWarning** method to get the additional **SQLWarning** objects, if any.
 - You can choose to ignore these warnings, since they do not stop execution with an exception.
 - The chain is cleared at the invocation of the next **Statement executeQuery** method or **ResultSet next** method.

Support for SQLWarning

- SQLWarning enjoys only spotty support.
 - Oracle only supports **SQLWarning** on scrollable result sets. The **SQLWarning** instance is created on the client.
 - Derby has support for **SQLWarning**. Aggregates like SUM raise a warning if NULL values are encountered during evaluation. Unsupported **ResultSet** types raise a warning.
 - MySQL does not document **SQLWarning** at all, and it appears that it does not generally support it.
 - PostgreSQL has support for **SQLWarning** on transactions and data truncation.

SQL Exceptions and Proper Cleanup

- The **Class.forName** method could throw **ClassNotFoundException**.
- JDBC objects could throw **SQLException** or one of its subclasses.
- **Connection**, **Statement**, and **ResultSet** objects represent external resources – they are not created inside the JVM.
- Thus Java garbage collection will not free these objects.
- They must instead be explicitly closed; each of these interfaces offers a **close** method.
 - Always call these methods in a **finally** block.
 - Failure to observe this requirement could result in the dreaded, "too many open cursors" message appearing in the DBA's log files and eventually cause the database to stop.
 - This would happen if your code threw an exception and the **Connection close** method was not called.
 - To close all three objects above, you will actually end up with a nested **try/catch/finally** as seen on the next page. There are several variations to do this.
 - Remember when coding in a finally block, it is even more important not to throw additional exceptions and to keep the block as short as possible.

SQL Exceptions and Proper Cleanup

```
//Prior code here enclosed in a try block
catch (ClassNotFoundException cnfe)
{
    System.err.println(cnfe.getMessage());
}
catch (SQLException se)
{
    // SQLException handler here
}
catch (Exception e)
{
    System.err.println(e.getMessage());
    e.printStackTrace();
}
finally
{
    try
    {
        if (rs != null) rs.close();
        if (stmt != null) stmt.close();
    }
    catch (SQLException se)
    {
        // SQLException handler here
    }
    finally
    {
        try
        {
            if (conn != null) conn.close();
        }
        catch (SQLException se)
        {
            // SQLException handler here
        }
    }
}
}
```

In this lab you will add exception handling to `DisplayEmployees` using methods available to `SQLException` and `SQLWarning`. Database resources will be closed in a **finally** block to guarantee their **close** methods will always be called.

Detailed instructions are contained in the Lab 2C write-up at the end of the chapter.

Suggested time: 30 minutes.

Evaluation Only

JDBC Wrapper Classes

- Wrapper classes have become common in enterprise applications to hide the JDBC details and simplify development.
 - Database connection information could be configured in an XML file, allowing easy deployment to a new server.
 - A robust error handler is mandatory in a mission critical application, yet we want it to be nearly invisible to the application.
 - Logging options are often configured externally and handled automatically by a wrapper class.
- We will be using a thin wrapper class called **DBUtil** to handle connections and exceptions and simplify the code we will be writing in the labs.
 - The preferences setup in Lab 2A allowed us to set up connection strings for a specific database of our choice. Now we can hide the connection process in a wrapper with a call to the **getConnection** method.
 - We will use the **close** method to close all JDBC objects such as the **Connection**, **Statement**, and **ResultSet** objects. This will allow us to remove exception handling associated with closing these objects and put them in **DBUtil**.
 - We have no wrapper for creating JDBC objects other than **Connection**. After all, we need to write some code! We will, however, use the **handleSQLException** method to streamline the formatting and printing of exception information.

```
try
{
    conn = DBUtil.getConnection();
    stmt = conn.createStatement();

    // Print first and last name where
    // the salary is less than $20,000
    rs = stmt.executeQuery(query);
    while (rs.next())
    {
        String first = rs.getString("firstname");
        String last = rs.getString("lastname");
        int salary = rs.getInt("salary");
        System.out.println(first + " " + last +
            ": " + salary);
    }
}
catch (SQLException se)
{
    System.err.println("SQL Exception in Salary");
    DBUtil.handleSQLException(se);
    se.printStackTrace();
}
catch (Exception e)
{
    System.err.println(e.getMessage());
    e.printStackTrace();
}

DBUtil.close(rs);
DBUtil.close(stmt);
DBUtil.close(conn);
```

In this lab you will modify **EmployeeReport** to use a wrapper class called **DBUtil** to handle connections and exceptions and simplify the code we will be writing in the remaining labs.

Detailed instructions are contained in the Lab 2D write-up at the end of the chapter.

Suggested time: 20 minutes.

Evaluation Only

SUMMARY

- The JDBC classes provide a means to access database information in a generic way once we setup the database connection.
- Setting up the database connection is one of the most error-prone parts of JDBC. Later we will see another solution using data sources.
- Mapping from SQL database types to JDBC types to native Java types requires attention to the possibilities of data truncation and **SQLWarnings**.
- Over the past two chapters we have viewed database programming from the point of view of both the DBA and Java programmer.
 - Separation of concerns is important to remember when creating the database and providing access to the developer.
 - While creation and administration may be left to the DBA, queries and updates will be the domain of the JDBC program.
- Handling **SQLWarning** and **SQLException** is a complicated and time-consuming task.
 - Over half of our program is involved in handling errors.
 - The old adage is if you have to do something more than once, write a program!
 - For the remaining chapters, we will encapsulate the connection and exception processing in **DBUtil**.