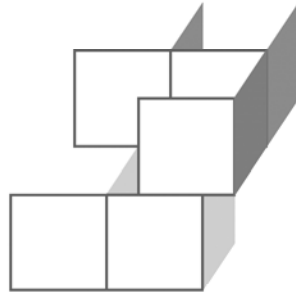
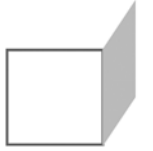




CHAPTER 2

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`
- 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 and 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.
- Use the JDBC 4.0 cause facility in `SQLException`

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.

JDBC 4.0

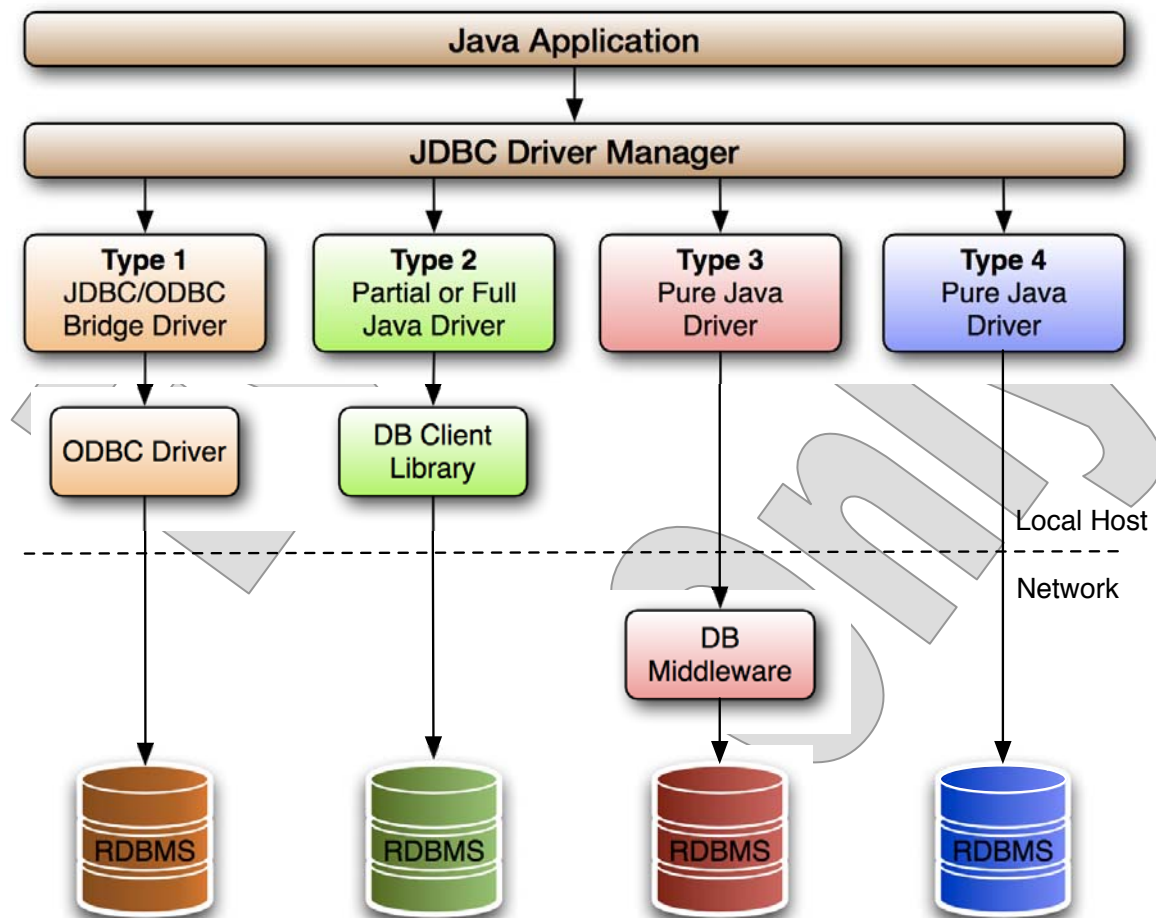
- JDBC 4.0 became a Final Release in December 2006, and is included with the Java 6.0 JDK.
- The **Service Provider** mechanism in Java 6.0 alleviates the need to explicitly load JDBC drivers.
- There is support for the **ROWID** SQL type using the **Rowid** interface.
- **SQLException** support has been updated.
- There is new support for SQL 2003 XML types.
- CLOB and BLOB support has been greatly enhanced, making it far easier to use these data types.
- Support for JDBC 4.0 requires Java 6.0 support as well. Some database vendors, notably Derby, provided JDBC 4.0 drivers shortly after its final release. Oracle started support of JDBC 4.0 with version 11 of the database, but the driver works with earlier releases.
- Java 6.0 includes **Java DB**, which is based on Derby 10.6 included with this course.

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, but had to support it again after customer outcry.
 - While IBM and Sun gave tacit support, only IBM implemented it in its products. It is not widely used today.
- **Enterprise Java Beans 2.1 (EJB) offers 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.
- **Java Persistence API (JPA) provides a higher-level abstraction of object persistence than JDBC, without being tied to an EJB container.**
 - EJB 3.0 entities use JPA for their implementation.
 - JPA is emerging as a standard in its own right, available to other containers and as part of Java SE.
 - Most implementations of JPA use JDBC.

JDBC Framework

- The **JDBC DriverManager** is implemented in the JDBC API included in the JDK.
- The drivers, written by the database vendors, 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.

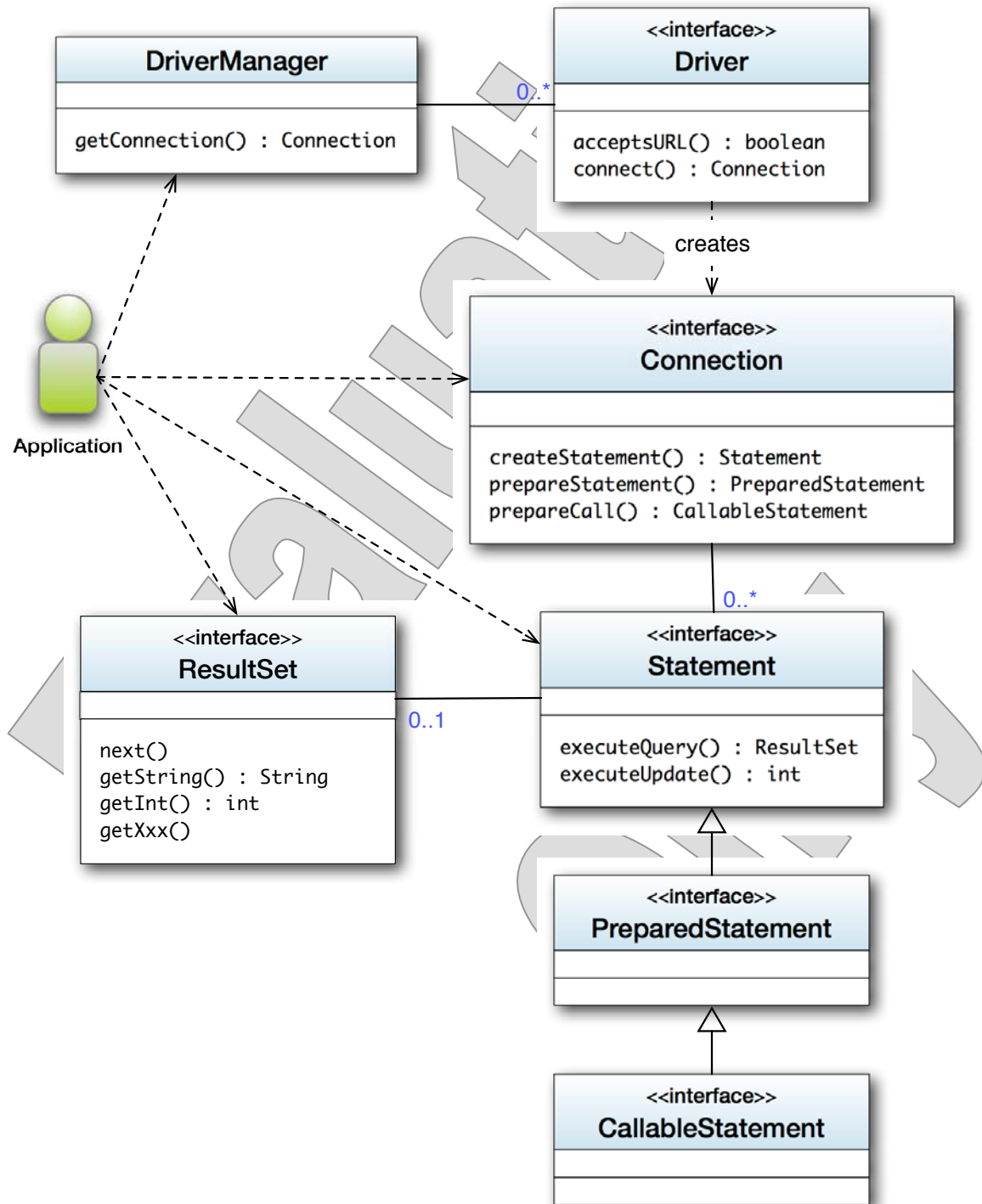


JDBC Drivers

- 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.

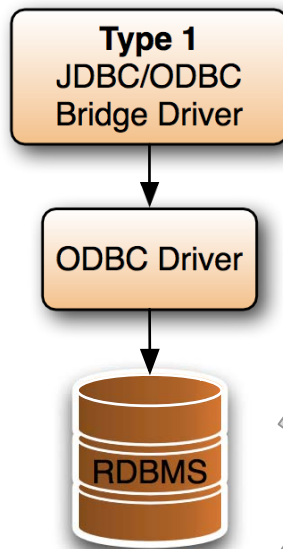
JDBC Interfaces

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



JDBC Driver Types: Type 1

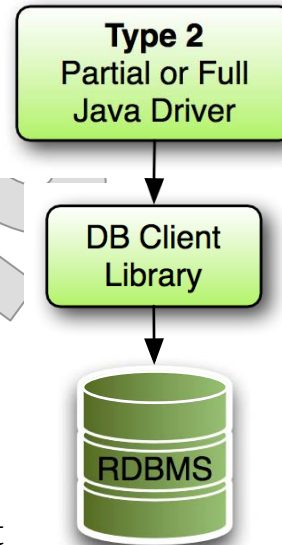
- There are four driver types defined in the JDBC specifications, known simply as types 1 through 4.
- Type 1 is a JDBC-to-native-API bridge.



- 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.

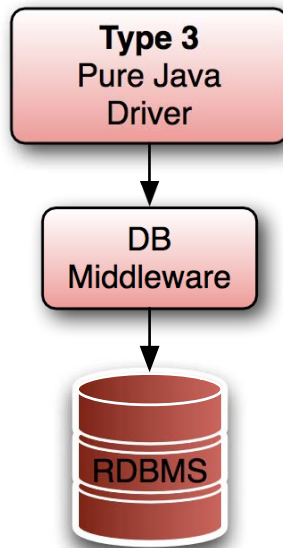
JDBC Driver Types: Type 2

- 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.
 - An example is Oracle's OCI (Oracle Call Interface) driver which communicates with the database. It is also known as the 'thick' driver.
 - 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.



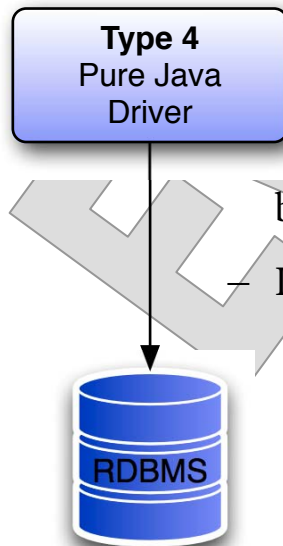
JDBC Driver Types: Types 3 and 4

- 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://developers.sun.com/product/jdbc/drivers`
 - This list is somewhat outdated.
- 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://db.apache.org/derby/derby_downloads.html`
 - Derby includes the driver in the same jar file with database engine. Derby was based on IBM Cloudscape.
- 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**.
- One **Connection** problem might be: “Class not found”. This message usually means the driver is not located on the CLASSPATH.
- 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 the scope of this course.
 - Under JDBC 4.0, create the text file **services.java.sql.Driver** to state the **class name of the desired driver**; place the file in the **META-INF** directory of a JAR or under any node of the class path.

Making the Connection

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

- The sample 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.
 - On the server side, data sources allow for encryption of username and password.
- 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:” or “jdbc:oracle”.
 - From there, every vendor has its 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://localhost/earthlings`

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

(**xe** in Oracle 10g Express)

Setting Database Preferences

LAB 2A

Suggested time: 10 minutes for Derby or 30 minutes for Oracle.

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 found at the end of the chapter.

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.ClientDriver");
private static final String DATABASE_URL =
    ("jdbc:derby://localhost/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.

Using Statements and ResultSets

LAB 2B

Suggested time: 30 minutes.

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

Detailed instructions are found at the end of the chapter.

Evaluation Only

SQL and Java Data Types

- The JDBC API defines many SQL types in **java.sql.Types**.
- Some common JDBC to java types mappings 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
ROWID	java.sql.RowId
SMALLINT	short
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
VARCHAR	String
XML	java.sql.SQLXML

- See Appendix C for a complete list.
- Two types were added to the table above in JDBC 4.0.
 - **java.sql.RowId** maps to the SQL ROWID values. Methods are added to several interfaces to allow access to this type.
 - **java.sql.SQLXML** maps as a reference to a SQL XML type.

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.
 - In SQL, mathematical operations with NULL values always return 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.
- **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.

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

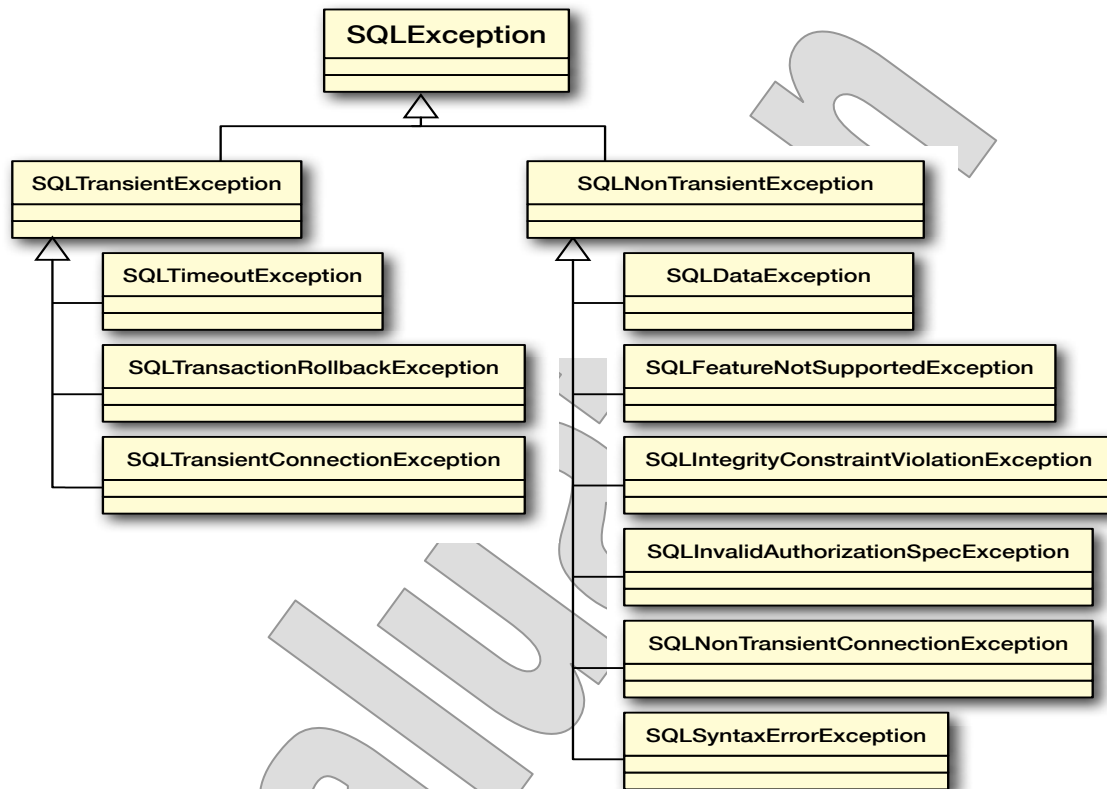
```
//Prior code here enclosed in a try block
catch (ClassNotFoundException cnfe)
{
    System.err.println(cnfe.getMessage());
}
catch (SQLException se)
{
    // SQLException handler here
}
catch (Exception e)
{
    System.err.println(e.getMessage());
    e.printStackTrace();
}
finally
{
    try
    {
        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
        }
    }
}
}
```

JDBC 4.0 Cause Facility

```
catch (SQLException se)
{
    while (se != null)
    {
        System.err.println(se.getSQLState());
        System.err.println(se.getErrorCode());
        System.err.println(se.getMessage());
        // Add JDBC 4.0 cause when known
        Throwable cause = se.getCause();
        while (cause != null)
        {
            System.err.println(cause);
            cause = cause.getCause();
        }
        se = se.getNextException();
    }
}
```

- The **cause** allows you to know the underlying reason, or cause, for the **SQLException**. For instance, an **IOException** throwing a **SQLException** would show up using the **getCause** method.
- New **SQLException** constructors were added to allow passing a reference to a cause for the exception.
- Causes can be nested, just like **SQLException**.
- In practice, the cause is often null or simply repeats the exception's error message.
- The JPA reference implementation, **EclipseLink**, makes extensive use of the cause facility, which aids in application debugging.

New JDBC 4.0 SQLException types



- The new **SQLException** classes were added to allow for more portable exception handling across database drivers.
 - It is not necessary to consult associated error codes or SQL states to handle transient and nontransient exceptions.
- **SQLTransientException** and its subclasses might succeed when retried without changing anything.
 - Possible problems could be timeouts and deadlocks.
- **SQLNonTransientException** and its subclasses will fail again until the underlying cause is corrected. Problems could be:
 - Coding, query, and constraint violation errors.
 - Data errors and failed connections.

Exception Handling

LAB 2C

Suggested time: 30 minutes.

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 found at the end of the chapter.

Evaluation Only

JDBC Wrapper Classes

- Wrapper classes have become common in enterprise applications to hide the JDBC details and simplify development.
 - A robust error handler is mandatory in a mission critical application, yet we want it to be invisible to the application.
 - Logging options are often configured externally and handled automatically by a wrapper class.
- JDBC 4.0 adds the notion of wrappers in **java.sql.Wrapper**, providing the retrieval of a proprietary delegate instance when the instance in question is in fact a proxy 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.

Example – Wrapper Classes

```
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);
```

Using a Wrapper Class

LAB 2D

Suggested time: 20 minutes.

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 found at the end of the chapter.

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**.

Setting Database Preferences

LAB 2A

In this lab you will edit database driver strings, test the database connection and insert these strings into user properties for future labs. It is mandatory that you complete this lab successfully to set up the preferences; otherwise none of the remaining labs will work correctly.

The files you will modify and test in this lab are used to assure that you can successfully connect to and communicate with the database. **DBPreferences.java** checks your connection strings and attempts to read from the schema you installed in the previous chapter. It uses **SetPrefsUtil.java**, which actually does most of the work.

SetPrefsUtil.java attempts to perform a query on the **employees** table, and returns an error if it fails. **SetPrefsUtil.java** has been ‘trained’ to recognize many error conditions reported by the supported drivers and will attempt to translate any of them to a more meaningful error message.

DBPreferences.java will also store your connection string information using the Java preferences API (**java.util.prefs**). This step allows you to set up these strings once in this lab and use them for the remainder of the course. You will find that the utility classes in later labs will take care of making the connection for you, based on these user preferences.

Lab workspace: Labs\Lab2A
Backup of starter code: Examples\DBPrefs
Files: src\cc\DBPreferences.java
src\cc\util\DBStrings.java
src\cc\util\SetPrefsUtil.java

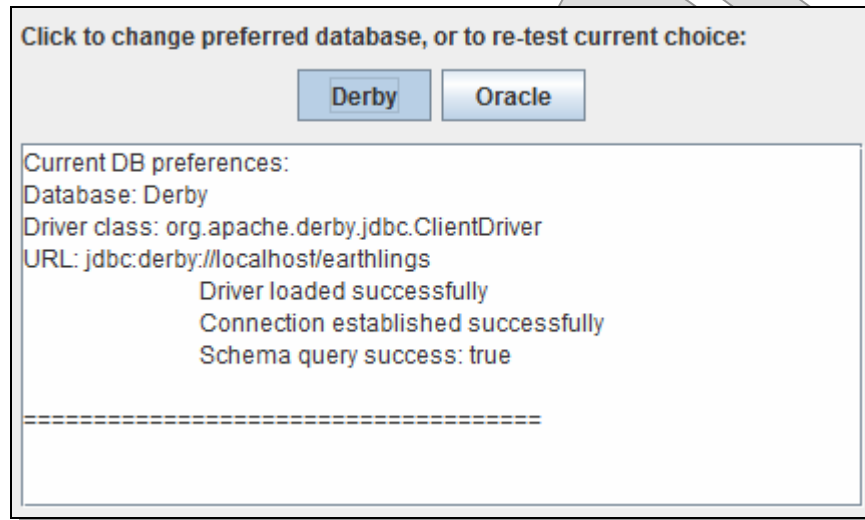
Instructions

1. Open **DBStrings.java** in an editor and notice the **TODO:** comment near the top. The connection strings for two databases, Oracle and Derby are already defined below it.
If you are using the Derby database shipped with the lab, then no modifications are necessary and you are ready to run the application.
2. The Oracle strings might need modification of the host, port, or SID. If you are using Derby, start the network server as indicated in the **Installing the Derby Schema** HTML file. If you are using Oracle, start the database according to their installation instructions; remember to start the database and for Oracle, and the listener as well.
3. If you are using Oracle, copy the Oracle driver to the **Capstone\Tools\Oracle\lib** directory. This driver should be the latest version of **ojdbc6.jar**, as the same driver will work with Oracle 9i, 10g and 11g. (The driver itself will work with Oracle 7.3.4 and later even though Oracle doesn't support versions earlier than 9i.)

Setting Database Preferences**LAB 2A**

4. Build the application using the script **build.bat** and then run **DBPreferences** using the script **run.bat**. Click the button corresponding to the target RDBMS, and you should see results similar to those shown below:

```
build
run
```



5. If there is an error, the application will attempt to return a meaningful solution to the problem in a line prefixed with "Solution" – as in:

```
Database: oracle.jdbc.driver.OracleDriver
      Driver loaded successfully
SQL Error
Solution: Oracle listener is running, but the database is stopped or
the SID does not exist
...
localhost:1521:orc
```

6. In the case above, the SID is incorrect and should be changed from "orc" to "orcl". Make the necessary changes and retest.
7. Another possibility is for you to encounter a problem with setting user preferences. On a Windows OS, the preferences API stores the preferences in the registry. This rarely causes a problem, but the Linux OS and Unix OS use the file system. Depending on system setup and file privileges, you could encounter an error such as:
- ```
java.lang.SecurityException: Could not lock System prefs. Lock file
access denied.
```
8. If you run into problems, let your instructor know, so corrective action can be taken. Changes to access rights will solve most preferences problems.

## Setting Database Preferences

## LAB 2A

9. You may close the application at this point, and in many cases this will be the last time you need to run it for the remainder of the class. However, if you like, you can experiment with the application at this point. It offers a few useful behaviors:
- You can toggle between supported databases as much as you like by clicking the buttons. If you have more than one of these RDBMS' available in class, you may want to leave this application up while running later labs, to test your code against, say, Derby and then Oracle.
  - You can click the currently selected button: this will not change user preferences but will re-test the database connection and show the results.
  - At each startup the application will check the current preferences and automatically test connectivity to that database.
  - There is also a command-line interface to the application:

**run test**

Tests current preferences and shows results.

**run set <database>**

Sets preferences to <database> and tests.

**run clear**

Clears user preferences.

**run**

Shows the GUI we've been working with so far.

## Using Statements and ResultSets

### LAB 2B

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.

|                                |                                   |
|--------------------------------|-----------------------------------|
| <b>Lab workspace:</b>          | <b>Labs\Lab2B</b>                 |
| <b>Backup of starter code:</b> | <b>Examples\Report\Step1</b>      |
| <b>Answer folder:</b>          | <b>Examples\Report\Step2</b>      |
| <b>Files:</b>                  | <b>src\cc\EmployeeReport.java</b> |

### Instructions

1. Open **EmployeeReport.java** and note the **TODO:** comments in the code. Over the next few steps you will insert new lines below these comments. When the new lines have been tested successfully, then remove the **TODO:** text, leaving the remainder of the comment to serve as internal documentation, if the text would prove useful. Use the page titled “Now All at Once” as a reference to complete your work.
2. The first step is to use the **Class.forName** method to load the driver.
3. Create a **Connection** object called **conn** using **DriverManager** to load the URL, user name, and password.
4. Create a **Statement** object called **stmt** using the **Connection** object, **conn**. You will notice the query string is already provided:

```
query = "select * from employees where salary < 20000"
```

This query will return a subset of employees whose salaries fall below \$20,000.

5. Create a **ResultSet** object called **rs** using the **Statement** object, **stmt**.
6. Create a **while** loop using the **next** method to work with the returned result set.
7. Inside the loop, create **Strings** called **first**, **last**, and an **int** called **salary**. Use the **getString** method or **getInt** method as necessary to print the results.
8. Build and test your application, using the **build** and **run** scripts. (These will be available for all exercises.) The results should look like this:

```
build
run
John Amdell: 15000
John Ayer: 15000
John Gus: 15000
Dona Leonard: 16000
David Santana: 19000
John Smith: 15000
Earlene Taylor: 18000
```

# Exception Handling

**LAB 2C**

In this lab you will add exception handling to **EmployeeReport** 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.

|                                |                                   |
|--------------------------------|-----------------------------------|
| <b>Lab workspace:</b>          | <b>Labs\Lab2C</b>                 |
| <b>Backup of starter code:</b> | <b>Examples\Report\Step3</b>      |
| <b>Answer folder:</b>          | <b>Examples\Report\Step4</b>      |
| <b>Files:</b>                  | <b>src\cc\EmployeeReport.java</b> |

## Instructions

1. Open **EmployeeReport.java** and see the **TODO:** comments to determine where to insert new lines. Use the page titled “SQL Exceptions and Proper Cleanup” as a reference to complete your work over the next few steps.
2. Create a query to display first name, last name, and job description for each employee whose salary is greater than \$90,000. It will be necessary to perform an inner join between the **employees** and **jobs** tables to do this. Refer to the previous chapter if necessary to construct this query.
3. Second, you need to check for any **SQLWarning** objects after the execution of the query in **stmt**. Use the code on the **SQLWarning** page as an example.
4. In addition to this, add exception handlers for main try block for **ClassNotFoundException**, **SQLException**, and **Exception**.
5. Next add a nested finally block to close the **ResultSet**, **Statement**, and **Connection** objects. Carefully examine the **TODO:** comments and their indentation to aid in creating the nested finally block correctly. You will be closing the **ResultSet** and **Statement** objects in the outer finally block. In the inner finally block, close the **Connection** object. The reason we are doing it this way is that we always want to close the connection regardless whether we are successful in closing the statement and result set. Unclosed connections could lead to catastrophic problems on the database with heavy use. Use the code on the page, “**Putting it all together: Adding error handling to the mix**” if necessary, but think through the process.

**Exception Handling****LAB 2C**

6. When you are done, build and test. Your output should look like:

```
Orval Johns, Database Administrator: 120000
John Bigboote, Vice President: 95000
Rebecca Zimmerman, President: 97000
```

7. Test your work by introducing errors in the **query** variable and formatting your exception output. An exception message should occur and look similar to:

```
SQL Exception in Salary
SQL State: 42X05
Vendor code: 30000
Message: Table 'EMPLOYEE' does not exist.
```

In the case above, the table **employees** was mistyped as **employee**.

## Using a Wrapper Class

LAB 2D

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.

**Lab workspace:** Labs\Lab2D  
**Backup of starter code:** Examples\Report\Step5  
**Answer folder:** Examples\Report\Step6  
**Files:** src\cc\EmployeeReport.java

### Instructions

1. Open **EmployeeReport.java** and see the **TODO:** comments to determine where to insert and delete lines.
2. Change the import from **java.util.prefs.Preferences** to **cc.util.DBUtil**.
3. Delete **Preferences** and **static final String** declarations, as we don't need them any more.
4. Change the lines shown in the source to eliminate a call to **Class.forName**.
5. Change the creation of **conn** from using **DriverManager.getConnection** to use **DBUtil.getConnection**.
6. Eliminate or change the exception handling as indicated in the code comments to take advantage of the error handling in **DBUtil**. You can choose to eliminate the **SQLWarning** block as well; we will not be using it in future labs.
7. Call **DBUtil.close** to close all database objects.
8. When you are done, your application should behave identically to the completed version from the previous lab – both for correct cases and exceptions.