



Capstone Courseware, LLC

33 Boylston Street
Jamaica Plain, MA 02130

877-227-2477
capstonecourseware.com

JDBC Programming

Edward Rayl
Cynthia Gustaff Rayl

Instructor's Guide

Revision 6.0



Revision Notes

Revision 6.0 updates the course to Java 6.0 and JDBC 4. Significant changes include:

- Supports/ uses Derby 10.4, MySQL 5.0 with 5.1.17 driver, Oracle 9/10/11, PostgreSQL 8.3.
- New treatment of the JDBC-4 cause facility.

Revision 5.0 overhauls the course for Java 5.0. Major changes include:

- Derby 10.1 is now included. Derby 10.0 did not support updatable result sets and this required a 'work-around' lab file and instructions for Derby. This lab and instructions have been eliminated.
- When using Java 5, two utility files supporting typed collections will be used, eliminating a warning that would have been generated otherwise.
- PostgreSQL 8.0 is now a supported database.
- The instructions for creating the course schema objects for each supported RDBMS are now in external XHTML files.

Revision 4.5.1 is a branch revision to add support for Derby 10.1 and PostgreSQL 8.0 while still targeting J2SE 1.4.

Revision 4.5 is the initial revision.





Course Overview and Philosophy

This course is intended for intermediate to advanced Java programmers. With a diversity of backgrounds in mind, the course has been designed to accommodate students with little or no database programming experience. The course has been designed to work with the following RDBMS':

Oracle versions 9i and 10g.

MySQL versions 4.1.x

Apache Derby 10.1 or later (IBM donated Cloudscape to Apache in August 2004, when it was renamed Derby).

PostgreSQL 8.0.

The RDBMS you or your client chooses must be installed and command line connectivity tested prior to course commencement.

We target a classroom environment that is stripped down to a J2SE SDK and a text editor. Many instructors will allow students to use an IDE as part of the training. We encourage this, and the layout of the lab software should make it very easy to wrap exercises in IDE projects. Specifically, this course has been tested on Eclipse 3.02. Lab files specific to Eclipse can be made available.

Chapter 1 provides a review of relational database and SQL fundamentals. It also contains the information necessary for installing the course RDBMS files. Please note the course assumes the RDBMS product has already been installed.

Chapter 2 goes through the basics of explaining the JDBC API, making a database connection using JDBC, and creating queries. Proper exception handling is stressed.

Chapter 3 covers scrollable and updatable result sets, prepared and callable statements, transaction support, batch processing, and other advanced topics.

Chapter 4 offers an introduction to row sets and their advantages over result sets. While all row set types are mentioned, only **CachedRowSet** is explored. A follow-on chapter will explore other row set types.





The Eclipse Overlays

Some students and instructors may prefer to install an IDE and use it in working through the course exercises. Capstone Courseware provides an optional package of workspace and project files for Eclipse 3.1 for this course. (See the course Setup Guide for download URLs.) Instructors, use this package on your own initiative and at your own risk. You should have experience yourself with Eclipse before using the overlay package in the classroom. The workspaces and projects have been tested lightly with the course but are not part of the standard product.

That said, these overlays should save a good deal of work for those who wish to use Eclipse instead of the text editor and command-line tools that are standard for the course. See the file `c:\Capstone\JDBC\Eclipse\ReadMe.html` for general notes on how to use the Eclipse overlays for Capstone courses. Be prepared to walk students through the first few exercises in Eclipse; the notes in this file are for experienced Eclipse users, and will not be clear to many students on their own.

One foible of Eclipse is that workspaces cannot be arbitrarily relocated without confusing the workbench a bit. To wit, if you choose not to install the workspace to the default, `c:\Capstone`, you will probably find that projects either don't build when they should, or that when you try to run them the Eclipse launcher "can't find the main class." The bottom line is the projects all need to be refreshed if they are opened from a path other than the one at which they were last open. So in this case the best process is to have everyone begin by opening, refreshing, and then closing all projects; this should clean up any odd behavior due to the "surprise" location.

The lab image and this IDE workspace come with support for Derby, MySQL, and PostgreSQL. To use the software with Oracle, you must have the appropriate JDBC driver JAR. The coursebook includes instructions for setting this up for use from the command line; to get labs working in the IDE with Oracle, do the following:

1. Choose **Window | Preferences** from the top-level menu.
2. Navigate the preferences tree to **Java / Build Path / User Libraries**.
3. Click **New ...**
4. Enter the name "Oracle" for the new user library, leave the "System Library" box unchecked, and click **OK**.
5. Select your new library ("Oracle") from the list, and click **Add JARs ...**
6. Navigate to your JDBC driver JAR -- typically this is `c:\Capstone\Tools\Oracle\lib\ojdbc6.jar` -- and click **Open**.
7. Click **OK** to close the Preferences dialog.



Again, Capstone Courseware can only offer complete technical support on the standard course, and while we hope this overlay is convenient, it is not as thoroughly tested as the core lab image at this time. If a given exercise is giving trouble, please be certain to build and run it from the command line, using the SDK tools as prescribed in the student guide, before contacting Capstone.

We always welcome feedback on our courseware, and especially with this new undertaking we would appreciate whatever comments and criticism you might have. Eclipse, like all IDEs, tries to promote ease of use by providing many different ways of doing the same thing; this is convenient for users but does leave a lot of questions as to what's best practice. At this time we believe consensus is still forming around a number of basic practices, and we'd like to hear how you use Eclipse for training situations and how this overlay works out for you. Please contact Will Provost at **provost@capstonecourseware.com**.





Timelines

Day 1

Chapter 1 SQL Fundamentals Review

Chapter 2 JDBC Fundamentals

Chapter 3 Advanced JDBC

Chapter 4 Introduction to Row Sets

As discussed above, different audiences will require different paces. If time is short, cut chapter 4. For programmers who have SQL skills, you can proceed directly to setting up the database schema in Chapter 1.





The RDBMS'

This course originally started life with HSQLDB as the embedded database of choice. IBM Cloudscape was released to open source in August 2004 as Apache Derby. Derby then seemed to be a better choice for several reasons. Derby has a far cleaner command line interface than HSQLDB. With Apache support behind it and dedication to SQL standards support, it appears to be a better choice than HSQLDB. Derby had better stored procedure support. In a test of CachedRowSet, Derby performed superbly where HSQLDB failed. Recently, HDQLDB has been incorporated into OpenOffice 2.0, and is known as OpenOffice Base.

Derby Jars:

derby.jar	Both the Derby database and JDBC driver
derbynet.jar	JDBC server to allow network connections to Derby from multiple clients
derbytools.jar	Derby command line utilities such as IJ
derbyprocs.jar	Contains Capstone specific Derby stored procedures - not a part of Derby

Based on the table above, the only jar that needs to be on the class path to compile and run the Java code is derby.jar. The Examples/SQL/derby and Demos/SQL/Derby directories require derbytools.jar and derbyprocs.jar in order to build the database. Nothing uses derbynet.jar at this point. It is included since it is part of the Derby distribution.

The embedded Derby database only allows for one connection to the database at a time. Closing the database connection is not sufficient to release a lock on the database, so it is necessary to close the database itself using:

```
getConnection(url + "; shutdown=true", username, password);
```

The **DBPreferences** application does this in **SetPrefsUtil** to allow its continued use in class along with an IDE for instance. The command line utility, **IJ**, does not do this, so the student will get connection errors if this utility is left running. This problem will be addressed in the future after the Apache Derby project releases an open source network driver.

MySQL has had its issues as well, some of which are documented in the course. With row set, it prematurely commits rows to the database in clear violation of the specifications. This cannot be rolled back either. When doing updates involving a join, it is actually possible to lose data. MySQL by default does not support transactions. An option to

```
CREATE TABLE with TYPE=INNODB
```





allows support for foreign keys and transactions, and is used in the DDL scripts in this course. Just the same, database constraints are ignored by MySQL. It is possible for the student to make serious mistakes with MySQL and never discover them. Be on guard! MySQL versions earlier than 4.1 should work fine with this course, but are not supported. Testing has been done with MySQL 3.23 and 4.0, however. MySQL does not support an identity column that starts with a default value other than one. Since the **employees** table starts employee numbers at 1001, a hack was introduced into "populate.sql" to get around this shortcoming. This particular hack did not work correctly with all versions of MySQL when used in the DDL script. Other databases might produce an error on this statement, but no harm comes of it.

The nature of the DDL scripts place a requirement on using Oracle 9i or later. Since a majority of corporate users have migrated to Oracle 9i or 10g, this should not pose a problem in most cases. If you are going to use an Oracle database for this course, be sure to read the Oracle installation guide cover to cover before proceeding. There are many pre-installation requirements, especially on Linux and Unix. Ignoring these cautions could lead to serious problems in the classroom.





Chapter 1

The instructor material for this chapter is rather extensive. This document assumes you are basically a Java programmer and might have a limited knowledge of SQL and relational databases. The extra information is in here so you'll be sure to cover completely at least those points critical for later chapters. That said, remember this chapter should be covered as quickly as possible so you can get to the JDBC meat in the balance of the book.

It is critical you know the password to the "SYSTEM" username if you are running this class with an Oracle database. The setup guide states the password should be "system", but it is possible for the Oracle installer to specify a different password. Make sure you talk to the Oracle installer and get the password for the "SYSTEM" username.

This chapter provides an overview of relational databases and SQL basics. Since one of the prerequisites to this course is a familiarity with basic SQL code, this chapter is meant to take only an hour to an hour and a half (including demos and lab). For those students desiring a little bit more about relational databases and entity relationship models, you can direct them to the appropriate appendix. Unfortunately, the time constraints of this course do not allow for any of this information to be covered during class time.

When you get to the section defining SQL, it would be helpful to make sure the students understand the issues raised due to various releases of the SQL standard; this will help reinforce the need for the "SQL Escape Syntax" section they'll be looking at in the "Advanced JDBC" chapter. As a review for you:

ANSI in the United States, and ISO internationally, publish the SQL standard. There have been several releases of the standard, each subsequent release providing additional functionality. Each of the following releases has two or more levels:

SQL86

SQL89

SQL92 (also known as SQL2)

SQL99 (also known as SQL3)

SQL2003

None of the database vendors fully implement all levels of the SQL92 standard, much less subsequent releases of the standard. In any case, SQL2003 is too new. Therefore, developing portable SQL code often requires foregoing the use of anything above SQL92 entry level, as well as vendor specific (i.e.: non-standard) SQL statements.

Also in chapter 1 there is a chart showing how SQL may be logically viewed in subsets. This course is only concerned with DML and TCL. DCL statements include "GRANT" and



"REVOKE". There are examples of "GRANT" in both createSchema4MySQL.sql and createSchema4Oracle.sql, both of which are scripts called in the first demo. Those two scripts and createSchema4Derby.sql have several examples of DDL statements.

There are two back-to-back demos in this chapter. The first demo directs you to refer the student to the xhtml file for the RDBMS you are using in this course. There is one for each of the RDBMS' supported in this course. The second demo is exactly the same for all of the databases.

The first demo installs the schema and opens a command console for the appropriate database. All the installs begin with a file "createdb.bat", that is different for each database. All "createdb.bat" files call a script, that actually creates the schema objects for each database. Only the Oracle create schema script calls an intermediate script. Each script is different because of the differences in DDL syntax for each database product. However, each of these scripts calls the exact same "populate.sql" script because the DML syntax is the same for all supported RDBMS'. **Do not edit "populate.sql"**. The demos and labs in **all** of the chapters are dependent on specific rows inserted into the schema. If this script is modified, then at the very least the output in the chapters will no longer match what you and the students will be seeing on the console. As discussed earlier, there is a MySQL hack in "populate.sql" that may generate warning messages in another RDBMS. See the section below on transactions for additional notes on the install schema scripts.

This chapter concerns itself only with DML syntax. If a student really wants to know anything about DDL syntax, for example how do you specify a column can or cannot accept NULL values, then refer the student to the scripts used in this first demo.

It is imperative the first demo be executed by you **with the students** performing the steps as you follow the instructions for the course's database. This demo familiarizes the students with the course database, the "home directory" for the course database and the tables that will be used. It installs those tables via script files and provides for a simple verification of the installation of the tables. It also gives you an opportunity to make sure each student knows how to open a command window in a specific directory - in this case the "home directory" for the course database. During later chapters the students will have to rebuild the database schemas, so the students will have to know how to do this demo eventually.

You will notice later, Lab 3B introduces callable statements, but MySQL 4.1 and below do not support stored programs. If you will be using MySQL, you may want to the students to install Derby as well, just so this lab can be completed later.

One other note about the Derby Demo: when using Derby interactively, failure to type the line,

```
set schema earthlings
```

as the first command after running the "ij" batch file could result in the "table does not exist" error.



For those students who are not very familiar with an RDBMS, it might be helpful to explain why joins are so common.

In a relational database, **normalization** is a series of steps designed to remove data redundancy (thereby reducing data inconsistency) by placing the data into multiple tables. A table is in **normal form** if it satisfies certain constraints. In many ways it can be considered formalized common sense.

E. F. Codd's original work identified three such constraints. There are now five generally accepted forms of normalization (plus a codicil to the third normal form). A discussion of the normal forms is outside the scope of the course. However, what's critical for the student to understand at this point is that the processes of placing the data into multiple tables works because the end result is tables connected via relationships. At the time of database build, each such relationship becomes a primary key / foreign key relationship in the resultant tables. In a nutshell, this is the reason why SQL queries with joins are quite common in the relational database world. As in Chapter 1 demos and lab, for example, you will often see something like a job id in a row that points to a job table that contains a row where you find the name and other information about a particular job.

Setting up foreign keys is part of DDL. If a student really wants to know about the syntax, again, refer the student to the scripts used in this first demo.

In the middle of the chapter there is a section on transactions. Spend time on this section. Later chapters in the course depend on the students understanding it. Keep in mind MySQL and PostgreSQL start with autocommit on. When using either of these RDBMS' you might have to remind the students to turn autocommit off when they open one of these SQL terminals.

When using MySQL entering

```
set autocommit=0
```

allows for transactions. The **set** is executed in MySQL's "createSchema4MySQL.sql" script. The students are also instructed to manually enter the **set** upon completion of the install schema scripts.

When using PostgreSQL entering

```
\set AUTOCOMMIT off
```

allows for transactions. The command is case sensitive. However, if the **set** is executed in PostgreSQL's "createSchema4PostgreSQL.sql" script, then the generic "populate.sql" script will abort on the MySQL hack mentioned above. So, the **set** is not executed here and students are instructed to manually enter the **set** upon completion of the install schema scripts. The down side to this is every commit statement in the generic "populate.sql" script will generate a warning message:



```
psql : populate.sql : 285: WARNING:  there is no transaction in progress
```

This warning, of course, may be ignored.

The chapter doesn't actually teach basic SQL, rather it talks about it and the second demo and the lab provide the student with examples of simple SQL queries. In order to ease students back into SQL code, the demo and the lab both start out with simple SQL statements that could be done more efficiently with an inner join. These two queries are followed by the inner join. The inner join uses ANSI SQL. Some students may be familiar with the following version of an inner join (see the "DML and Transactions" demo, step 5):

```
select employees.id, first_name, last_name, job_id, state
from employees, jobs
where jobs.id = job_id
and job_name = 'Trainer';
```

The lab deliberately gives the student the code for solving the problems. Again, this is because this chapter is only meant to be a refresher on writing SQL code. The only time answers are not given in the lab is when the student is repeating a SQL statement used earlier in the lab.





Chapter 2

This is the pivotal chapter in this course. You should expect to spend one hour on lecture and one and one half hours on labs. Later chapters depend on a strong understanding of this chapter, so spend the necessary time. If you do end up spending more time than expected, then Chapter 4 might need to become optional.

A lot of thought went into getting the student to be successful from the beginning in communicating with a database. This turned out to be a large chore, since several RDBMS' are supported and a simple mistake anywhere along the way will usually cause the whole program to fail. Since most mistakes are made getting the connection strings right for the database, the first lab concentrates on these strings. The utility class, **DBPrefs**, attempts to diagnose common mistakes. Once the strings are properly adjusted, the **DBPreferences** GUI is used to test the connection to the database and ascertain the schema has been properly installed. A support class, **SetPrefsUtil**, puts the strings into the user's Java preferences for later use in the labs. **DBPreferences** proves its worth because it allows the student to change to a different database quickly and test again. Be sure to become familiar with this utility and its utility classes so you can help debug the inevitable problems that will show up in class. As an aside, **DBPreferences** was built in the Eclipse Visual Editor (VE 1.02) and will display properly in that IDE. For JBuilder and JDeveloper, `initialize()` would have to be changed to `jbInit()` at the minimum. Testing in these IDEs was not done however.

This chapter starts by covering the JDBC 3.0 API and the packages that contain it. It is probably a good idea to show the students the Javadocs for `java.sql` and `javax.sql` so they are able to locate the various interfaces, classes, and methods as they are covered in the text. Besides, the Oracle SQLJ compiler changed every bit of SQLJ to JDBC.

The SQLJ API was avoided in this course due to its low adoption. While several books have been written on the subject, most are aging at this time. It is interesting to note Oracle dropped support for SQLJ, and disabled the SQLJ translator in the 10g database. Customer outcry has forced them to relent and it was re-enabled in 10.1.0.4 patch set. Just the same, it is a subject in and of itself. JDO 2.0 failed to garner support, but finally succeeded in a JSR 243 "Public Review Reconsideration Ballot" in February 2005.

The "JDBC Interfaces" UML class diagram underwent several adjustments during the writing of this course. You are likely to see several variations on interface associations if you see this class diagram in other texts. In particular, the diagram shows a dependency between **Driver** and **Connection**. It could be argued this is an association, since the **DriverManager** factory is responsible for creation of the **Connection** object. On the other hand, it could be argued the diagram gives the student the most accurate information on the API, from the perspective of using it in application code. The diagram implies loose relationships from **DriverManager** to **Connection** but tight ones from there through to **ResultSet**, and this seems to reinforce the rest of the chapter's lessons about managing **Connection**, **Statement**, and **ResultSet**. An actor is used in this class diagram to show the user typical interface usage from an application. A '0..1' association is shown between



ResultSet and **Statement**. While a **Statement** can create many **ResultSet** objects, opening a new **ResultSet** implicitly closes any other **ResultSet** objects. It was therefore decided not to use '0..*'.

The various JDBC driver types are discussed in some detail. It is interesting to note that no database vendor other than Sybase has a Type 3 driver. It was thought type 3 might become popular as a means to allow access to legacy data stores in addition to the corporate database. The problem was none of these solutions layered a transaction manager on top of the implementation. We might prefer use JTA, dedicated hub-and-spoke integration, a MOM solution, etc. It appears type 3 has fallen out of favor with most developers. In the case of the type 4 driver, it has been highly optimized by both Oracle and IBM. For instance, the Oracle type 2 driver uses SQL*Net for the protocol between the driver and the database. The Oracle type 4 driver does not to use it, electing instead to use the sub-protocol of SQL*Net itself. This is the very reason why the Oracle type 4 driver is often (but not always) faster than the type 2. IBM has followed a similar pattern. Neither of these vendors offers a type 3 driver.

Class.forName() is considered to be the preferred method for loading drivers. It allows dynamic loading of the driver based on a **String** representation of the fully qualified class name. As such, the database **String** can be externalized to an initialization file to avoid hard coding it into the application. You can read more about **Class.forName()** at:

<http://www.javageeks.com/Papers/ClassForName/ClassForName.pdf>

While modern drivers initialize and register themselves with **DriverManager** in a static block, older drivers did not necessarily do this. In that case, you would use **Class.forName("driverclassname").newInstance()**. There are situations in Derby where you might use this construct as well because of its embedded nature. Use it if you get the error, "no suitable driver" when trying to close an existing connection and then reopen it.

Lab 2B assumes the student has gotten past the ugly part of creating a successful connection to the database. While this lab uses dubious means for closing database objects and exception handling, this was done to simplify the code. Every effort is made to make the student successful in the first lab where code writing is required. Every line the student needs to write has a nearly identical example at the top of pages in the text, the only difference being the table name. The SQL query string is supplied for this lab to allow the student to concentrate on JDBC code.

Data type conversion is introduced. You should use the tables in Appendix C with this topic and go over several scenarios with the student. For instance, you might explain when it would make sense to retrieve a number as a **String**. The tables in Appendix C are more complete than the table in Chapter 2. Additionally the appendix tables can serve as tear-outs for the student.

Exception handling in JDBC is often done incorrectly, even by experienced Java programmers. The student might have been exposed to some bad habits already, so be prepared to explain this to their satisfaction. The concept that outside resources such as



database objects, are not cleaned up by the garbage collector is not apparent to all students. Some may not really understand the need or use of a **finally** block. If they get this wrong, the cost could be high in an enterprise application. Have them chant this for you, "Always clean up database objects in a finally block."

At the end of this chapter, database wrappers are mentioned. In order to keep all labs as concise as possible, a wrapper class, **DBUtil**, is introduced to handle connections, closing database objects, and exception handling. This wrapper should be viewed as a convenience class only. It had to stay thin to allow the student to write code for the fundamentals in later labs. Please explain to them the simple purpose of **DBUtil**. It is not to be used as an example of best practices. The student should welcome this class after having to complete Lab 2C. It eliminates about 100 lines of code for each succeeding lab.





Chapter 3

This chapter builds on the foundation laid out in the previous chapter. This is also the chapter where you can experience differences in behavior between drivers. Even though all drivers may say they meet the JDBC 3.0 specifications, there is at least enough vagueness in the specification to allow for differences in implementation. Outstanding issues present in driver/database combinations supported in this course will be mentioned in the text.

You should be able to deliver the lecture in less than an hour and the labs should take about an hour and a half. If timing becomes an issue please remember Chapter 4 could be considered optional, as mentioned above. In addition, batch processing could be skipped without impacting student understanding of Chapter 4.

The chapter starts out by introducing **executeUpdate()**. This method can be a Swiss army knife for the Java programmer who is not afraid of SQL. You might want to expand on this page and introduce the student to some of its usefulness.

Scrollable result sets were introduced in JDBC 2.0 after substantial demand. Just the same, Oracle PL/SQL programmers are still writing millions of lines of code without this functionality. For all its appeal, it might lack justification. Updatable result sets can be very useful, and support for it is increasing.

Lab 3A might initially appear more complex than necessary. The problem requires queries on two tables **and** updates at the same time. This lab demonstrates to the students how to query the database and make changes based upon the values in the query results.

With close examination of Lab 3A, you may notice it can be made more efficient. For instance, a subquery would be more efficient, as it would not require a full table scan of employees. But the nested code would still be required because for each employee this application needs the values of `minimum_salary` and `maximum_salary` from the jobs table to perform the updates. This lab would not be any simpler (in fact it would be slightly more complex), and it would be require an explanation of subqueries as well. It was decided not to do this. Just the same, if you were to do this, the query would be:

```
String queryEmp = "select id, firstname, lastname, salary, job_id " +  
    "from employees where job_id = " +  
    "(select jobs.id from jobs where job_id = jobs.id " +  
    "and (salary < minimum_salary or salary > maximum_salary))";
```

This works properly in Oracle, MySQL, PostgreSQL and Derby. An update on a join will produce an error in Oracle, PostgreSQL and Derby. In MySQL, updates on a join are unpredictable and best avoided.

The SQL escape syntax is often tough for the student who encounters it for the first time. Examples of usage seem to do the best job. Dates and timestamps are notorious for having proprietary versions. For instance, a default DATE in Oracle takes the form, YY-



MON-DD, as in 05-MAY-21. This can be changed at the session or database level with **NLS_DATE_FORMAT**. Obviously the Java programmer would prefer not to know all this, and at the same time keep the code database independent.

When getting to prepared statements, there is one thing that is not immediately obvious to the student. It is easy to get 'lost' in usage of single and double quotes in a query string as used in a **Statement**. Bind parameters in **PreparedStatement** can actually make this problem easier and many programmers may choose to use a prepared statement just for this reason. While we usually preach that **PreparedStatement** is faster than **Statement** if the statement is executed several times, this isn't necessarily true. On some databases, the breakeven point might be 100 iterations or higher.

CallableStatement is only supported on Oracle, PostgreSQL and Derby. MySQL will be adding it for version 5. The function used for this course creates an email name by concatenating the first initial of the first name to the last name and then truncating it to eight characters or less. While this could be done more easily in Java, the point really is one of maintenance or responsibility. It is quite common, for instance, to have complex reports written as a database procedure. If it works well, the DBA is very unlikely to allow the Java programmer to recode it. The old adage, "if it ain't broke, don't fix it" often applies. The stored function, **build_email**, is inserted in the database when it is created. You can see this code in the createSchema4Xxx.sql scripts for Derby, PostgreSQL and Oracle. Additionally, if you look in the Examples\StoredFunction directory, you would see the source for them. In the case of Derby, Derby.java is compiled and put in derbyprocs.jar before inserting into the database. If you look in createSchema4Derby.sql, you will find you can test build_email by uncommenting the second line below:

```
-- Test the function
-- values earthlings.build_email (' John' , ' Smith' );
```

If your students understood the discussion on transactions in Chapter 1, then your job will be easier when we revisit them here from the Java perspective. **Savepoints** are fairly new and need to be tested for support. Every database has slightly different support for transaction isolation levels. It is important to point this out when covering this topic. Oracle has gone to great lengths to avoid table or row locks because of speed issues, and might use snapshot data to achieve isolation. Therefore, be careful if you decide to explain the implementation of each isolation level.

Batch processing can be skipped, as mentioned above, if time is short. It is important to let the student know commercial database vendors have spent a lot of time and money perfecting data loading tools. Java programmers implementing batch processing will often be reinventing the wheel, and probably won't do the job as well.





Chapter 4

Rowset provides many advantages over **ResultSet**, and this optional chapter introduces **CachedRowSet** in the lab. A future chapter will cover the remaining row set types. You should expect lecture to take about 30 minutes or less, and the lab will take about 30 minutes.

If you are working in a J2SE 1.4 environment, then the RowSet implementation is not included with either Java or the lab files. The installation person should have installed it, but you should probably verify this installation. You can read more about the details at:

<http://www.capstonecourseware.com/Setup/rowset.html>

The implementation of **RowSet** as defined in JSR 114 and included in Java 5.0 is relatively new. At the time of this writing, no vendors have implemented JSR 114 natively in their drivers. Even the Oracle 10g **RowSet** implementation is proprietary. We depend on Sun's reference implementation: **com.sun.rowset**. Using this implementation with the existing drivers was a challenge due to bugs either in **com.sun.rowset** or the vendor's driver. The **CachedRowSet** used in Lab 4 is carefully crafted to avoid these problems in the supported databases. It is quite likely some students may encounter these bugs while completing the lab. Be prepared to examine their work in light of the solution file. If any students receive a **SyncProviderException**, have them rebuild the database and try again. This problem happens intermittently, but a rebuild has always eliminated the exception.

The service provider interface and event notifications sections can be skipped if necessary. Neither subject is used in the labs.





Errata

At this time there are no errata for the course.





Feedback

We truly do welcome feedback, both of a specific nature (pointing out mistakes) and general suggestions. For the former sending email with a numbered list of corrections would be most helpful.

Please send feedback to:

Will Provost
Capstone Courseware
<mailto:provost@capstonecourseware.com>
www.capstonecourseware.com

