



**Capstone Courseware, LLC**

33 Boylston Street  
Jamaica Plain, MA 02130

877-227-2477  
capstonecourseware.com

# Advanced Java Programming

**Will Provost**

*Instructor's Guide*

**Revision 1.4**



## Revision Notes

**Revision 1.4** is the initial revision, targeting the J2SE 1.4 SDK.





## Course Overview and Philosophy

What is “Advanced” Java? The student guide asks this same question, in the course overview. There are many different views on this: which topics are properly considered introductory, intermediate, advanced? Which are fundamental to Java development, and which are optional?

The course overview (the six presentation pages prior to the detailed tables of contents) provides a good summary of this, and since you will probably need to begin your class by discussing this exact question with students it is placed there, in the student guide, to aid that introductory lecture. The gist is that we have two goals for this course: learn a few fundamental-yet-advanced topics like threads and serialization; and learn how to build enterprise applications using J2SE features.

This latter idea is on the one hand a strong organizing principle, we think: from Module 2 through Module 4, there is a high-level story to the course, as you work from the presentation tier back to the persistence tier. On the other hand, this philosophy will doubtless have its detractors: after all, doesn't the 'E' in J2EE stand for 'enterprise?' The first and foremost answer is that we're not promoting J2SE over J2EE – only saying that J2SE has a number of its own strong answers to the multi-tier architectural question, and that some or all of these can play a role in real, well-considered Java development. In fact there can be some mixing and matching: if students are interested, you might take some time to talk about JFC clients making Web requests, or RMI as the basis for EJB transport (and mention the RMI/IIOP and JAX-RPC subsets), and JDBC as part of servlet-based multi-tier applications as well as entity beans.

And each of these technologies can be used for many purposes, “enterprise” or multi-tier architecture or no. You will most likely encounter the greatest interest in either JFC or JDBC, while RMI has become something of a dark horse with the maturing of J2EE. The course is truly modular at this level, so you can feel free to devote more time to these topics at the expense of RMI. But especially for those interested in distributed Java systems, some treatment of RMI will doubtless provide some good perspective on how those systems work – issues of managing remote references, remote vs. serializable, distributed garbage collection, and others being general, even if RMI solves them in specific ways.





## Timelines

### Day 1

Module 1, Chapter 1 Threads  
Module 1, Chapter 2 Reflection  
Module 1, Chapter 3 Serialization

### Day 2

Module 1, Chapter 4 Sockets  
Module 2, Chapter 1 Introduction to JFC  
Module 2, Chapter 2 Basic JFC Application Design

### Day 3

Module 2, Chapter 3 Swing Components  
Module 2, Chapter 4 Architectural Patterns  
Module 3, Chapter 1 RMI Architecture

### Day 4

Module 3, Chapter 2 RMI Implementation  
Module 3, Chapter 3 Practical RMI  
Module 4, Chapter 1 SQL Fundamentals Review

### Day 5

Module 4, Chapter 2 JDBC Fundamentals  
Module 4, Chapter 3 Advanced JDBC  
Module 4, Chapter 4 Introduction to Row Sets

If time is tight, there are really very few chapters in this course that are absolutely necessary – perhaps none. Rather, there are a few dependencies, such that if X is not covered, it will be hard to cover Y. Let student interest guide you if any cutting is necessary. For starters, the final chapter in each modules is designed to be optional. Nothing will depend on it. Chapters on Reflection and Serialization are strongly recommended, and the RMI module will refer to these APIs. JFC is used in both the RMI and JDBC modules, but all JFC code in those modules is provided, so there isn't a strong dependency there. Of course, the SQL Fundamentals chapter can be skipped for a SQL-savvy audience.





## 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 3.0.2 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\JavaAdv\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.

For this course, some specific notes:

- Many of the JFC applications load images, and they do so by looking for image files as class resources. The primary lab image holds these files in the class path under **c:\Capstone\Classes**, so the Eclipse projects have an **Images** folder linked to that location, and their classpaths include GIF files from there. This should be transparent in practice, but students may wonder how the image loading succeeds when running from this different class path.





- In the RMI module, the relationship between the RMI registry and the individual projects is problematic for Eclipse. The command-line build scripts direct all classes to a common **TARGETPATH**, and this is on the user's classpath, so the registry can be run from anywhere and will always see the latest class files. But Eclipse insists on keeping compiler output separate, from one project to the next, and none of its **bin** directories will be on the classpath. To solve this, an additional source folder **src** is added to every project, with a single class **StartRegistry**. Run this instead of using the command **rmiregistry**; it just creates a default registry, but automatically inherits the output path of the project from which it is run. This allows it to see the server's stubs and thus the server and client can interact through it. You will have to shut this registry down and start a new one for each project – largely true for command-line users, too.
- The Weather case study in the RMI module relies on a number of resources, for servers and clients: forecast HTML fragments, city and state lists, the US map. The primary lab image holds these files in the class path under **c:\Capstone\Classes**, so the Eclipse projects have a **Resources** folder linked to that location, and their classpaths include GIF files from there. This should be transparent in practice, but students may wonder how the image loading succeeds when running from this different class path.

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](mailto:provost@capstonecourseware.com).





## Module 1 – Advanced Java

### Chapter 1:

Threads really are a basic feature of the Java language. However, most programmers won't use threads consciously for at least a year into full-time Java programming, either because they're writing things that don't use multithreading at all or because they move on to J2EE, where for component-building multithreading is discouraged or forbidden. Still, it's good to understand how threads function in the JVM; if nothing else to know that they're there, understand how the garbage collector is working, and to know where those exception stack traces come from! So this chapter starts almost from scratch, introducing threading concepts as well as the Java threading API. In this one chapter we just get a taste of the larger API; there is certainly room to expand here if students are interested.

### Chapter 2:

Again, in one chapter we can't go too deep into the Reflection API; we could spend a day on this for an audience of specialists. This chapter concentrates on the role of meta-data in the Java runtime, its exposure to application code through the API, and the ability to read meta-data from an application. The last section introduces the powerful concept that reflection can be used to make things happen, and not just to read about them and perform diagnostics. Here again one could go further if students are interested.

### Chapter 3:

This chapter introduces Object Serialization as the logical extension of data formatting streams. Note that our Courses 103 and 104, which are listed as prerequisites for this course, end on this very same chapter. So students may have seen it already. We include it in this course with the expectation that many audiences will come to the course from Java experience or training other than our Courses 103 and 104, and may not know serialization even though they have strong Java skills in general. Understanding of Java streams, however, is a true prerequisite, as stated in the course outline. If students don't yet know much about streams, a little review on the whiteboard will be time well spent.

Concentrate on the tremendous flexibility of the serialization engine: how completely generic, yet how powerful and simple to use. We note the basic pitfalls, in particular re-initialization of transient fields on object reads, but the chapter does not try to go deep on issues like versioning or externalization.





## Chapter 4:

This is the first of two APIs that connect Java code to standards outside the Java architecture – the second being JDBC, with its relationship to SQL. Here we must first understand the OSI/RM and “what’s a socket?” before teaching the API. There’s no need to go much deeper than the presentation material in this first section, unless students are curious; but it sets up key concepts and terms including TCP and UDP. Then we pursue two common goals for sockets programming: connected conversation over TCP – specifically HTTP, which is so very common – and connectionless communication using UDP and datagrams. The labs should give both a “cookbook” sense of how to get started with a sockets programming task and an introduction to the sorts of processing problems that are common in sockets coding: timing sends and receives based on content length or an end-of-transmission byte, blocking vs. non-blocking, etc.





## Module 2 – JFC

### Chapter 1:

This chapter sets the stage for the rest of the module. It introduces the Java Foundation Classes (JFC), a collection of APIs that allows developers to build full-featured GUI applications, and its major parts: AWT and Swing. The chapter may also serve as a brief introduction to AWT, covering the Component-Container pattern, layout management, and the AWT event model.

### Chapter 2:

In this chapter students get their first detailed look at Swing. Here the students build their first simple application that has all basic JFC features. The chapter presents the architecture of the Swing top-level containers. The Root Pane - Content Pane hierarchy is presented in detail.

### Chapter 3:

This chapter is the real flesh of the module. Here we learn how to use the most common Swing components. The chapter covers buttons, menus, and text components. Assuming there is enough time, the instructor may want to dig deeper into the Swing text management framework before Lab 3A and into data transfer before Lab 3B.

### Chapter 4:

Here we add some theory. This chapter introduces some of the primary Design Patterns used in JFC. We look how the Design Patterns are applied to the Swing components design, and to application designs that use Swing. Lab 4A is also a good point to talk about the Swing components as Java Beans.





## Module 3 – RMI

### Chapter 1:

This is an entirely theoretical chapter, with no demos or labs. This lecture can be fairly quick, or, depending on students' interest and the instructor's enthusiasm for whiteboard-based expansion, it can go a couple hours. The point is just (a) to get RMI placed in the broader context of Java development and (b) to lay out the basics of the RMI architecture, placing some blank boxes that can be filled in progressively over the second and third chapters.

### Chapter 2:

Most of this chapter focuses on implementation practice, by way of exercises in a simple RMI application that shows the user interactive weather forecasting. Most of the code is on the client side, actually (this code was borrowed from our JavaBeans course); one of the client GUI components has a little bit of code to look up an RMI service and call its one method to get HTML content. Students will work along the traditional path from interface to implementation class to server implementation, and then will learn how to get RMI software up and running with the help of the RMI registry.

The later part of the chapter allows students to experiment a bit with true remote invocation, pairing off to allow for a server machine and a client machine. This should give them a clear understanding of the limited fileset necessary for functioning of an RMI client, as they get rid of all the server class files except the interface and stub and see the client continue to function. If a network is not available, it is possible still to illustrate the proper code deployment by having client and server run based on two different class paths on one machine. Create a `c:\Capstone\ClientClasses` directory next to the existing classes, and build the system to both targets. Then set different class paths in two different consoles: one that starts the registry and server, and one that runs the client. Then the lab instructions to remove server-side files for the client class path will make sense and give the desired results. (Again, with Eclipse this exercise will have to take an entirely different shape, perhaps involving new projects that include subsets of the original project code.)





## Chapter 3:

In this chapter you will look more closely at a number of elements that have been taken for granted in the previous chapter. The lab software uses a new application for this chapter that is a bit more complex, with several remote interfaces, a server and three different clients, one of which also implements and passes a remote-object subclass. Stub and skeleton code is shown as a way of leading students into a deeper consideration of the practical issues of getting method invocations over the wire and back. (RMI can be so effortless that it sometimes takes a conscious effort to remember that what's going on under the hood is decidedly non-trivial.) The rules for parameter passing are laid out, and this in turn leads to a discussion of the classic issues of design for latency: use of remote references as opposed to serialized objects in this case. The Value Object pattern is introduced and the second lab takes students through several experiments converting use of remote references to use of serialized objects.

If there is extra time, a wrap-up discussion would be well advised. Especially it can be an opportunity to get students thinking about the RMI features that haven't yet been discussed:

- Remote class loading
- RMI security
- RMI proxy servers (just as a design pattern)
- Distributed Garbage Collection
- Multiple remote interfaces per implementation class
- The RMI activation model
- RMI/IIOP

Finally, it might be worthwhile to discuss the changes to RMI included with Java 5.0 – especially the dynamic loading of stubs, which makes RMIC obsolete and simplifies development and deployment considerably. The next revision of this course will target the J2SE 5.0 JDK across the board, and so this among other changes in APIs will be internalized.





## Module 4 – JDBC

### Environment and Setup:

We've tried to make the course software as trouble-free as possible. We have also tried to preserve portability of the lab image away from the default installation root of **c:\Capstone**. If it's possible to install to this default location, we recommend that you stick with it, as it will simplify various tasks for students; however various parts of the code and scripts refer to a single environment variable **Capstone\_HOME**, and this can be set differently at the beginning of the class to reflect a different root location.

If a student neglects to set **Capstone\_HOME** as described in Chapter 1, the first few labs will work anyway in most cases; the scripts in them have been written to make a best guess as to the correct value of **Capstone\_HOME**, and to set it themselves. In fact it's possible to run through the entire course in a single DOS box without setting **Capstone\_HOME** explicitly, because early exercises will set it for the console and later ones will find it. But it is unlikely that every student in the class will stick with a single console for the whole day! Hence the instructions in Chapter 1.

### Chapter 1:

The instructor material for this chapter is rather extensive. That's because this document assumes you are basically a Java programmer and have might 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 that are critical for later chapters. That said, remember that this chapter should be covered as quickly as possible so you can get to the JDBC meat in the balance of the book.

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 is written out three times, one for each of the databases supported in this course; typically you will only walk students through one of these, although using Derby along with one of the other two databases might be interesting for comparison of certain features – just keep an eye on the clock! 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 three installs begin with a file **createdb.bat**, which is different for each database. All **createdb.bat** files call a script, which 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 same **populate.sql** script because the DML syntax is the same for all three databases.

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.

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 that a column can or cannot accept NULL values, then refer the student to the scripts used in this first demo.

It is imperative that 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.

You will notice later that 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 both MySQL and Derby, so this lab can be completed later. If for any reason a student has to rebuild the Derby schema at any time during the course, they'll already be familiar with this demo and (hopefully) you won't have to stop the flow of the class to go over this.

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.

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.

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, firstname, lastname,
       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.

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 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. 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.

## 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 is the best option for cutting.

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 three databases 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, which are captured in the **cc.util.DBStrings** class. The **cc.DBPreferences** utility attempts to diagnose common mistakes. Once the strings are properly adjusted, the GUI is used to test the connection to the database and ascertain that the schema has been properly installed. **DBPreferences** proves its worth because it allows the student to change to a different database quickly and test again – especially useful if you are running against multiple databases, e.g. MySQL and Derby. Be sure to become familiar with this utility so you can help debug the inevitable problems that will show up in class.

Note that **DBStrings** is replicated throughout the tree of exercise code; but this does not mean that students will need to change it everywhere if they change it once! A support class, **cc.util.SetPrefsUtil**, puts the strings into the user's Java preferences, for use in later labs, whenever the user clicks one of the buttons in the GUI. All the other lab exercises start out by reading these preferences strings, and only use **DBStrings** for default values. So as long as Lab 2A is carried out completely, user preferences should be available and the values defined in that lab will be used the rest of the day.

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 `jblInit()` 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 student the Javadoc for **java.sql** and **javax.sql** so that they are able to locate the various interfaces, classes, and methods as they are covered the text.

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 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. A few notes in particular:

- The diagram shows a dependency between **Driver** and **Connection**. It could be argued that that this is an association, since the **DriverManager** factory is responsible for creation of the **Connection** object. We choose a simple dependency to orient the diagram as much as possible to the perspective of using JDBC it in application code: i.e. there may be deeper relationships between the driver manager, driver, and connection objects, but from the application programmer's perspective all that is implied is a dependency.
- 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 use 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. Again, emphasizing the application programmer's perspective on the API, we decided not to use 0..\*, which might be more precise from a driver programmer's standpoint.

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 that Type 3 might become popular as a means to allow access to legacy data stores in addition to the corporate database. The problem was that none of these solutions layered a transaction manager on top of the implementation. We might prefer to use JTA, dedicated hub-and-spoke integration, a MOM solution, etc. It appears that Type 3 has fallen out of favor with most developers.

The Type 4 driver 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 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 driver 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 that the student has gotten past the potentially 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 on the first lab where coding is required. Every line that the student need to write has a nearly identical example at the top of prior pages, 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 in this chapter. If there is time and interest, use the tables in Appendix C with this topic and go over several scenarios with the students. For instance, you might explain when it would make sense to retrieve a number as a **String**.

Exception handling in JDBC is often done incorrectly, even 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, it the cost could be high in a enterprise application, even if it is nil in their standalone-application labs. Have them chant, "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 that 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 that Chapter 4 could be considered optional, as mentioned above. As an alternative, batch processing could be skipped without impacting student understanding of Chapter 4, thereby allowing coverage of rowsets.

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 in justification. Updatable result sets can be very useful, and support for it is increasing. As mentioned earlier, Derby does not yet support updatable result sets. Lab 3A has a version for Derby that avoids the use of updatable result sets, but at a cost of more program complexity.

Lab 3A might initially appear more complex than necessary. The complexity here is that 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, you may notice that the code could 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 this application needs the values of `minimum_salary` and `maximum_salary` from the `jobs` table for each employee to perform the updates. This lab would not be any simpler (in fact it would be slightly more complex), and it would be require and explanation subqueries as well. It was decided not to do this. For reference, 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 and Derby, so students can feel free to try it out. However an update on a join will produce an error in Oracle and Derby, and 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 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 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 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 and Oracle. Additionally, if you look in the Examples\StoredFunction directory, you would see the source for both. 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 that 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 that 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. You should expect lecture to take about 30 minutes or less, and the lab will take about 30 minutes.



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 in **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 that some students may encounter these bugs while completing the lab. Be prepared to examine their work in light of the solution file.

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



## RDBMS' Supported for the Course

### Derby

This JDBC module 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 has better stored procedure support. In a test of **CachedRowSet**, Derby performs superbly where HSQLDB failed. At the present time, Derby does not support updatable result sets. This will be remedied in a future update to Derby. 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 gets an open source network driver.

### MySQL

MySQL has had its issues as well, some of which are documented the course. With row sets, 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.

## Oracle

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.

It is critical that you know the password to the "SYSTEM" username if you are running this class with an Oracle database. The setup guide states that 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.





## Errata

Since the most recent release of the course, the following issues have been discovered. These will be fixed in the next course release.

- Chapter 1, pages 15-16: the diagrams showing the two strategies of extending **Thread** or implementing **Runnable** are backwards. The diagram on page 15 should be on page 16, and vice-versa.





## Troubleshooting and Tool Tips

First, be sure to be familiar with the lab and environment setup instructions in the Table of Contents and the course Setup Guide. Assure that the instructions in the Setup Guide have been followed, and that you know the locations of the installed tools, passwords, etc. If, those steps having been followed, and a classroom machine is failing to run demos or lab answers correctly, here is a list of items to consider and to double-check:

- In the JDBC module, especially for Labs 2B and later, be sure that the **Capstone\_HOME** environment variable is set correctly. Remember (see "Environment and Setup" section earlier in this document) that the first demos and labs 1 and 2A will often run without incident even if this variable has not been set; after that it is necessary. Walk students through setting the variable globally (e.g. through the Control Panel's Setup applet) if needed.
- 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. This is a common mistake.
- Working with MySQL and Oracle, students can leave their SQL terminals running while building and testing lab code; but Derby in embedded mode and the **ij** terminal application will not support this. So another common problem will be students trying to test a lab with **ij** still running. The likely symptom is the following output from the Java application:

```
Failed to start database
'C:\Capstone\JDBC\Demos\SQL\derby\..\..\..\JDBC/Database/earthli
ngs', see the next exception for details.

SQL Exception: Failed to start database
'C:\Capstone\JDBC\Demos\SQL\derby\..\..\..\JDBC/Database/earthli
ngs', see the next exception for details.
```





## 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](http://www.capstonecourseware.com)

