DB2 Information Management Software

DB2 for z/OS and OS/390: Ready for Java

Setting up your Java-DB2 environment

Easy-to-use examples, including using SQLJ with WSAD V5.1

Java-DB2 usage hints and tips

Bart Steegmans
Ulrich Gehlert
Judy Ruby-Brown
Paul Tainsh

Redbooks

ibm.com/redbooks

International Technical Support Organization

DB2 for z/OS and OS/390: Ready for Java

December 2003

SG24-6435-00

Note: Before using this information and the product it supports, read the information in “Notices” on page xix.

First Edition (December 2003)

This edition applies to Version 7 of IBM DATABASE 2 Universal Database Server for z/OS and OS/390 (DB2 for z/OS and OS/390 Version 7), Program Number 5675-DB2 and UDB for Linux, Unix and Windows V8 FixPak 2, and WebSphere Studio Application Developer (WSAD) V5.1 (pre-GA Build).

Contents

Figures

Tables

Examples

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®	IBM eServer™	QMF™
CICS®	IBM®	RACF®
Cloudscape™	ibm.com®	Redbooks™
DB2 Universal Database™	IMS™	Redbooks(logo)™
DB2 Connect™	iSeries™	S/390®
DB2®	Language Environment®	SAA®
Domino™	MVS™	System/390®
DRDA®	MVS/ESA™	WebSphere®
e(logo)server™	Notes®	z/OS™
Encina®	OS/2®	zSeries®
eServer™	OS/390®	

The following terms are trademarks of International Business Machines Corporation and Rational Software Corporation, in the United States, other countries or both:

Rational®

The following terms are trademarks of other companies:

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

Preface

Read the following statements:

- The earth is flat!
- The earth is the center of the universe!
- Men shall never fly!
- Java will never work properly on the mainframe!

All four statements had a lot of advocates for a long time, but all of them turned out to be wrong. Completely wrong.

In this IBM® Redbook we show how Java and DB2® for z/OS™ and OS/390® can work together and form a strong combination that can run your mission-critical enterprise applications. This publication focusses on the new IBM Universal Driver for SQLJ and JDBC, IBM's new JDBC driver implementation, supporting both Type 2 and Type 4 driver connectivity to the members of the DB2 family, including DB2 for z/OS, and DB2 for Linux, Unix and Windows.

This publication provides guidance on the different ways to set up your environment to hook a Java program up to a DB2 for z/OS subsystem, through JDBC or SQLJ, using the Type 2 driver and the Type 4 driver.

We provide an SQLJ tutorial, and demonstrate how to develop and deploy SQLJ programs using the new SQLJ support functions that became available with WebSphere® Studio Application Developer.

We demonstrate the use of Java and DB2 using native Java programs, as well as through the use of Servlets and JSPs running on a WebSphere Application Server.

Who should read this publication

If you are a seasoned mainframe developer, this publication will help you understand that Java is now a first-class member of the programming language portfolio on the mainframe. You will find that developing in Java for the z/OS platform combines the best of two worlds: The performance and reliability of the mainframe with the sophisticated development tools on the workstation. You will also find that Java is the ideal development environment for enabling your DB2 system on z/OS for the World Wide Web.

If, on the other hand, you are an experienced Java programmer with no z/OS background, this publication will show you that the mainframe is as good a platform for running your applications as any other. Also, this publication has a strong emphasis on SQLJ, which currently does not have as much attention in the Java community as it should have.

The first part of the publication is an introduction to the Java programming language and environment, and is intended for people that are really new to Java. If you have prior experience with Java, you may wish to skip this part, or just review Chapter 3, "Accessing DB2 from Java" on page 23, which covers the JDBC API.

In the second part, we cover the installation process, describing which software you are going to need on both the mainframe and the development workstation, and how to set up connectivity from the workstation to your DB2 subsystem on z/OS.

Part 3 is the heart and soul of this publication. It demonstrates how to develop DB2 Java applications on the workstation and how to deploy and run them on the mainframe. We cover stand-alone applications using JDBC and SQLJ, and provide a tutorial and reference to SQLJ syntax and usage.

Finally, Part 4 talks about applications running in an application server environment, using the Servlet and JSP technologies.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

Bart Steegmans is a DB2 Product Support Specialist from IBM Belgium currently on assignment at the ITSO in San Jose. He has over 14 years of experience in DB2. Before joining IBM in 1997, Bart worked as a DB2 system administrator at a banking and insurance group. His areas of expertise include DB2 performance, database administration, and backup and recovery.

Ulrich Gehlert is an IT Architect with IBM Global Services, Germany. He has worked at IBM for seven years. His areas of expertise include the Java platform, with a strong focus on database and Web application server technologies, and the OS/390 and z/OS operating systems. He holds a degree in Computing Science from the University of Erlangen, Germany.

Judy Ruby-Brown is a Consulting IT DB2 Specialist in the United States. She has supported DB2 for OS/390 and z/OS for fifteen years in the IBM Dallas Systems Center (in the IBM Americas Technical Sales Support Organization). Her areas of expertise include parallel sysplex and DB2 data sharing, disaster recovery, and high availability. She has presented on these topics at the International DB2 Users Group (IDUG) in the US, Europe, and Asia Pacific, as well as at the DB2 Technical Conferences and SHARE. She holds a degree in Mathematics from the University of Oklahoma. She previously co-authored the *SAP R/3 and DB2 for OS/390 Disaster Recovery* publication.

Paul Tainsh is a DB2 Database Administrator with IBM Global Services Australia. He has 14 years of experience in DB2, as a developer, DBA and instructor. He has been working with computers since 1980 and has worked with mainframes since 1982, in both Australia and England. He previously wrote sections of *DB2 UDB for z/OS and OS/390 Version 7 Presentation Guide* for the ITSO.

Figure 1 Paul Tainsh, Bart Steegmans, Judy Ruby-Brown, Ulrich Gehlert

Thanks to the following people for their contributions to this project:

Julie Czubik
The editor who edited this publication
International Technical Support Organization, Poughkeepsie Center

Rich Conway
Bob Haimowitz
International Technical Support Organization, Poughkeepsie Center

Sigi Bigelis
Bill Bireley
Delmar Blevins
Curt Cotner
Heather Lamb
Peter Miller
Becky Nin
Paul Ostler
Peggy Rader
Tom Toomire
John Vonau
IBM Silicon Valley Laboratory, San Jose

Connie Tsui
IBM Toronto Lab, Canada

Ulrich Kawald
Dresdner Bank, Frankfurt am Main, Germany

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

`ibm.com/redbooks/residencies.html`

Comments welcome

Your comments are important to us!

We want our Redbooks™ to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  `ibm.com/redbooks`

- Send your comments in an Internet note to:

  `redbook@us.ibm.com`

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. QXXE Building 80-E2
  650 Harry Road
  San Jose, California 95120-6099

Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition may also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-6435-00
for DB2 for z/OS and OS/390: Ready for Java
as created or updated on December 22, 2004.

December 2004

This revision reflects the addition, deletion, or modification of new and changed information described below.

New information

- Information about the no-charge feature IBM z/OS Application Connectivity to DB2 for OS/390 and z/OS was added. See 3.2.6, “IBM z/OS Application Connectivity to DB2 for z/OS and OS/390” on page 34.

Changed information

- Added a shaded box on page 28 to clarify that the Universal Driver code that ships with DB2 for Linux, Unix and Windows should not be used on the z/OS platform. You should use the Universal Driver that ships with DB2 for z/OS V7 or V8 on the z/OS platform. This is true for both type 2 and type 4 connectivity. (You can of course use the Universal Driver that comes with DB2 for Linux, Unix and Windows *to access* data that resides in DB2 for z/OS (V6), V7 or V8 system, either directly via type 4 connectivity or via DB2 connect).

Part 1

Introduction

In this part we give a brief overview of the Java history, the different components that make up the Java technology, and how Java can be used to access a DB2 for z/Os and OS/390 system.

1

A brief history of Java

In its brief lifetime Java has become a significant force in computer application development, moving quickly from its initial function as a controller of small consumer devices to a language that runs on mainframes, and is a cornerstone of Internet application development. With this growth Java has expanded from being just a computer programming language to a complete specification of technologies to provide complete enterprise solutions.

This chapter describes:

- The development of Java as a programming language, and its evolution into a platform for providing enterprise solutions culminating in the J2EE specification.
- The creation of the JDBC and SQLJ specification for accessing relational databases, with an emphasis on DB2 for OS/390 and z/OS.
- The implementation of Java on IBM's OS/390 and z/OS platforms, and the effect this has had on the architecture of the mainframe.
- A few of the products developed to host and manage applications on this platform, specifically with their support of DB2 in mind.

1.1 From toaster to the enterprise

"*I dunno...the Internet?*" - Homer J. Simpson

1.1.1 Java and the consumer electronics revolution

Java started life as a problem to be solved. The developers of Java at Sun Microsystems saw that the next big thing in the evolution of computing was the rise of consumer electronics, and the increased computing power within these devices. They saw a move away from the desktop computer, with its proprietary operating system, to a multitude of consumer computers running video home recorders, sound systems, and even toasters.

The people at Sun Microsystems then looked at what was then one of the dominant programming languages, C++, and found it was not up to the task. With consumer devices, reliability, not speed, was seen as the key attribute. C++ with its heritage from C, was a mixture of object-oriented design with the procedural elements of C. Although C++ was a very powerful language, which could compile to very fast executable code, with pointers and such, it was also error prone. The computer users of the day were quite used to rebooting their desktop machines when fixing programming problems. This was not a situation that should occur on a remote control unit or a toaster.

Along with the issue of reliability, C++ tended to be platform specific. Developers had to be aware of what machine and what operating system they were coding for. If a program had to run on another platform, the program had to be recoded to run elsewhere. With the plethora of new devices you could not have this luxury. *Develop once, run many* was the mantra of this era.

So the new language was to use the good features of C++, while removing the areas that created potential for disasters. The new language was to be fully object oriented, and therefore based on reusable classes and objects. At the same time it had to be less complex, by removing the difficult areas in the old languages. The new language would not use pointers and multiple inheritance. It would handle the cleaning up of memory allocation requests internally, and not rely on the programmer to code for it. Memory leakage, where programmers request memory in a program, but not free it when finished with it, is a problem in many programs, and adds to the instability of these programs.

Along with this, the new language had to generate code that was truly reusable. It needed the ability to be run on all platforms with no change, from Windows or Unix machines, to small microprocessor-based machines with no operating systems at all.

In the early 1990's work was started on a new language called *Oak,* which was renamed to the more marketable name *Java* in January 1995.

Technology was possibly not the most significant reason for the acceptance of Java. Throughout its development there was a perceived need, that to achieve a critical mass of acceptance, the whole architecture and all its elements would have to be supplied to the developers for free across the emerging Internet.

Java was first released by Sun Microsystems in 1995. What even surprised the developers, was the speed at which it was accepted amongst the development community. Along with this acceptance came a flurry of activity to develop new functionality. What was perhaps most surprising was that this development was not just coming from the lone computer developer, the usual supporter of open systems, but also from industry giants such as IBM. These companies were involved in the process, from working on the base specifications with Sun, and releasing Java-based products to plug into their existing product line.

These additions were brought together in the release in 1998 of Java 2. As with most stages in the development of Java, this was only a step forward and not an end in itself.

1.1.2 Accessing the data

Over time, the development of Java was extensive, with the development of language specifications and extensive class libraries to expand on multiplatform functionality. The development in this area that is of most significance to us, is the development and enhancement of the Java Database Connectivity (JDBC) API specification. Originally, database access was not a significant aspect in the development of Java applications, as it was focused on small consumer products. With the rise of Java as a primary Internet development tool, the links to data increased, and the need for standard cross-platform database calls became necessary. Over time, the functionality of the JDBC API has increased dramatically as has the improvement in the physical code to drive these calls (the JDBC drivers).

1.1.3 The rise of the Internet server

As Java gained acceptance within the development community, some work had to be done to move from a development platform for small consumer devices, to one which would fit into and fulfill the needs of the large system development community. This move was given further impetus through the rise of the Internet and the business community's acceptance of this new technology as a tool of its trade.

The business technology community was going through revolutionary changes with the move away from the single platform model, which handled everything from the user interface, processing of business logic, to data access. Replacing this was a whole new platform relationship, where all these processes could be split off on to any number of platforms. Where there was once a mainframe connected to dumb terminals, all within the offices of the company, the new technology saw end users running browsers, which connected to business logic on middleware, connected to databases running on other server platforms. All this could be connected with a raft of new technologies such as message-oriented middleware, transaction monitoring, and object request brokers.

While this was happening outside, the Java platform itself was moving, with the addition of new features, such as Java Servlet support, to provide a simple method of removing business logic from front-end on to Java servers. An initial limitation of the stand-alone Java application and the Web-based Java applets was that they were both executed on the client machine and were limited to client-based resources. As you could never be sure what resources the client had, the client-based applications tended to go for the lowest common denominator solution, which assumed minimal client access to resources. This lead to static Web pages with limited appeal.

Java architecture was on the move towards server-based solutions, which, as they were executed on the server, opened up the browser to the power and resources that servers can provide. Business logic was removed from the client side to the server. The browser would handle the user interaction, and pass any requests to the server application. This application would contain all the business logic and database calls. The application would then pass the results back to the browser in the form of HTML to be displayed.

One of the first solutions outside Java to achieve this was through the use of a *Common Gateway Interface* or *CGI*. This solution always had performance issues. Every time a CGI application is executed, a new process is started on the host server. The process of starting and stopping CGI applications is very expensive to the host machine. If a lot of users are requesting the same process, this can lead to large bottlenecks on the host machine. Another limitation of CGI is that it is hard to link between processes once they are started, making it difficult to handle requests such as session authorization and logging.

The Java Servlet was introduced in 1997 to provide a Java-based solution for server-side processing. A feature of the Servlet is that it provides answers to the execution issues of using CGI. Servlets are executed in a shared Java Virtual Machine on the server, which is started once, and can then be reused for new requests. On the initial execution of a Servlet, the server initializes the Servlet once and then pools it in memory, making it readily available for new incoming requests. This cuts the cost of starting and stopping a Servlet. Additionally, Java Servlets are written in the Java language, and are able to use all the features of that language, such as platform-independent coding, in contrast to CGI.

At the same time the development of standard APIs for database access was going on. This culminated in the Java Database Connectivity (JDBC) API specification, and acceptance of the Enterprise JavaBean component architecture, which encapsulates business functionality into a easily configured and deployed components.

An issue also arose with Java Servlets. It had nothing to do with their performance this time. The difficulty arose from the fact that Web page design was rightfully moving from software developer's hands into the realm of graphic designers. The Servlet programming model does not support this "separation of concerns". The next step in the development of Java was to split these areas of responsibility, through the introduction of the JavaServer Pages (JSP) specification. This model allows Web designers to control and concentrate on the Web design, and the code developers to provide the designers the resources needed to create these pages while still using the performance and system management benefits of the Java Servlet.

Basically a JavaServer Page is an HTML page that contains Java code to execute program logic to generate dynamic Web pages. This design allows Web designers to build the HTML to control layout, while the developer adds the necessary logic needed to run the page, logically splitting the page development tasks.

A key feature of the JavaServer Page is that it is an extension of the Java Servlet, not a replacement. In fact, whenever a JavaServer Page is executed, it is first dynamically compiled into a Java Servlet, and it is this code that is run on the server.

1.1.4 The Internet and the enterprise

The needs of the large systems development community to manage the new server-based application architecture, and the vendor community to offer products to fulfill this architecture, along with the development of new Java components, culminated in the development of the Java 2 Technology Enterprise Edition (J2EE) standard.

The focus of this standard is to support a middle tier where the business logic runs. It defines a minimum level of support for the components that are required to achieve this.

We cover the J2EE technology in more detail in Chapter 2, "An introduction to Java 2 Enterprise Edition" on page 9.

1.2 Java and the z/OS and OS/390 platforms

One of the questions asked when porting the original Java reference implementation to OS/390, was whether a technology targeted at the consumer device and browser market would be viable on a mainframe server. The answer was yes with some reasonable modifications and additions.

Java was first made available with a port of the AIX® Java Development Kit 1.0.2, directly to OS/390 Unix System Services, in 1996. This was followed by the first fully supported release of JDK 1.1.1 for OS/390 in 1997.

Some of the initial issues to be handled were purely platform related. These included:

- Java technology is purely Unicode based, whereas the mainframe uses EBCDIC.
- Floating point data types that are used in Java were not implemented on OS/390 (Java mandates the IEEE floating point standard, while the S/390® hardware implemented its own proprietary format).
- Security in Java was based on “layers of defence”, while mainframe security is principal-based access control.

The solution to these issues were:

- Manage the differences between the mainframe architecture and the Java specifications, so while Java uses Unicode internally it was only converted to EBCDIC when the data was externalized. Also enable mainframe software products (such as DB2, for example) to handle Unicode data.
- Change the OS/390 hardware to support IEEE floating point.
- Extend the IBM development kit to make use of mainframe facilities, such as a adding interfaces to support principal-based security.

These changes were implemented across releases of the IBM development kit and IBM hardware.

A major concern with running Java on the mainframe was the performance of the JVM. Transaction processing on the z/OS and OS/390 platforms is characterized by short, repetitive transactions, that are run in subsystems such as CICS® and DB2. These environments focus on sharing resources across activities, by loading system-wide resources at subsystem startup time and then sharing resources across transactions, thus minimizing transaction startup and completion.

Compare this with the initial JVM port in 1996, where activity would kick off a full copy of the JVM, reloading all classes, every time. To illustrate the cost of starting a JVM, each initialization the JVM would load sixty system classes, allocate over 1000 non-array objects, and use over 700 array objects with a majority of these objects not being used by the application being executed.

The latest release of IBM’s Java Development Kit, now called IBM Software Development Kit (SDK) for z/OS, Java 2 Technology Edition V1.4, introduces persistent reusable Java Virtual Machines. This technology decreases transaction startup time by creating a common system heap of system classes and other sharable classes. This heap is created at subsystem startup time, and is then shared across all JVMs running in that subsystem. Each transaction running in this subsystem has its own JVM, which ensures isolation between transactions. However, when a transaction completes, the JVM is not destroyed but waits to be reused by the next transaction. By minimizing the startup cost by sharing system resources and reusing JVMs, Java technology is reaping the benefits of the mainframe platform.

You can get specific details about the Persistent Reusable JVM in the publication *New IBM Technology featuring Persistent Reusable Machines*, SC34-6034**,** which is available at:

http://wcs.haw.ibm.com/servers/eserver/zseries/software/java/pdf/jtc0a100.pdf

Along with the development of the base JDK, has been the continued development and enhancement of drivers for DB2 for z/OS and OS/390, which support the JDBC API specification. Initially, drivers were provided for Version 5 of DB2, which were reliant on the

ODBC specification, a database connectivity API that has been developed for the Windows platform but also implemented in DB2 for OS/390. Since then, the move has been to remove the need for ODBC, and provide drivers that have native support for DB2, culminating in a Java-only driver with DRDA® calls into DB2, allowing database access across multiple platforms.

As well as changes to the drivers, DB2 has been expanded to utilize the new technologies by providing services such as unicode support, IEEE floating point data types, and Java support in stored procedures and user-defined functions.

Java application support on OS/390 and z/OS

In its brief life as a mainframe language, Java has made huge inroads in that platform. Java programs can be run in the Unix System Services environment of OS/390 and z/OS, in exactly the same way as they are run under any Unix system. They can also be executed using JCL in the MVS™ batch environment (using a Unix System Services utility program), and they can be used to implement CICS and IMS™ transactions.

Perhaps the most significant area of support for Java programs on OS/390 and z/OS is provided by WebSphere Application Server running on the mainframe platform, which provides full J2EE support, allowing the execution of Java elements such as JavaServer Pages and Enterprise Java Beans. This publication covers the creation, installation and actual execution of some of these elements, and how they interact with DB2 for OS/390 or z/OS.

2

An introduction to Java 2 Enterprise Edition

The Java 2 Enterprise Edition (J2EE) is a specification that defines the set of technologies and their usage as building blocks to develop across enterprise applications.

This chapter discusses:

- The three different technology editions for the Java 2 platform: Java 2 Platform, Enterprise Edition; Java 2 Platform, Standard Edition (J2SE); and Java 2 Platform, Micro Edition (J2ME)
- The J2EE specification and the technologies that are defined within, including the JDBC specification used to access DB2 for OS/390 and z/OS databases
- The features of the Java 2 programming language, looking at issues such as object oriented coding, the significance of Java bytecode, and the Java Virtual Machine
- The different ways in which Java applications can be designed, developed, and executed, by looking at the Java application components

2.1 The three technology editions for the Java 2 platform

The Java 2 platform has been separated into three editions. Each of these editions focuses on a specific architecture and contains specifications for the technologies required to fulfill these. The three editions are:

- Java 2 Platform, Micro Edition (J2ME)

  Defines the technologies required for the highly optimized environment of consumer products, such as mobile phones, car navigation systems and pagers.

- Java 2 Platform, Standard Edition (J2SE)

  Provides the base environment for developing and executing Java applications and applets. It includes the Java Development Kit and Java Runtime Environment. By its nature, it is focused on development of code that is executed client side.

- Java 2 Platform, Enterprise Edition (J2EE)

  Focuses on the development and execution of server-based applications through the specification of technologies such as Enterprise Java Beans (EJBs), Servlets, JavaServer Pages, and JDBC. It is important to note that J2EE includes the J2SE specification. Due to the nature of development of applications using DB2 for OS/390 and z/OS, we focus on this edition within this publication.

2.2 Java 2 Platform, Enterprise Edition

Java 2 Platform, Enterprise Edition (J2EE) is a specification of the different technologies that are available to developers of enterprise Java applications. Typically these applications are server based, and may even include multiple servers of different types. This server focus makes it the architecture of interest for DB2 for OS/390 and z/OS developers.

It is important to note that the specifications that are contained in this edition are not solutions in themselves, but are just definitions of how the solution will be interfaced with, and how it will behave. Thus a solution, to be judged as compliant, must fulfill the functionality specified in the architecture, using the objects and methods defined within it.

The J2EE technologies

Figure 2-1 on page 11 gives an overview of the J2EE application model. We will now briefly discuss some of the technologies that are part of J2EE.

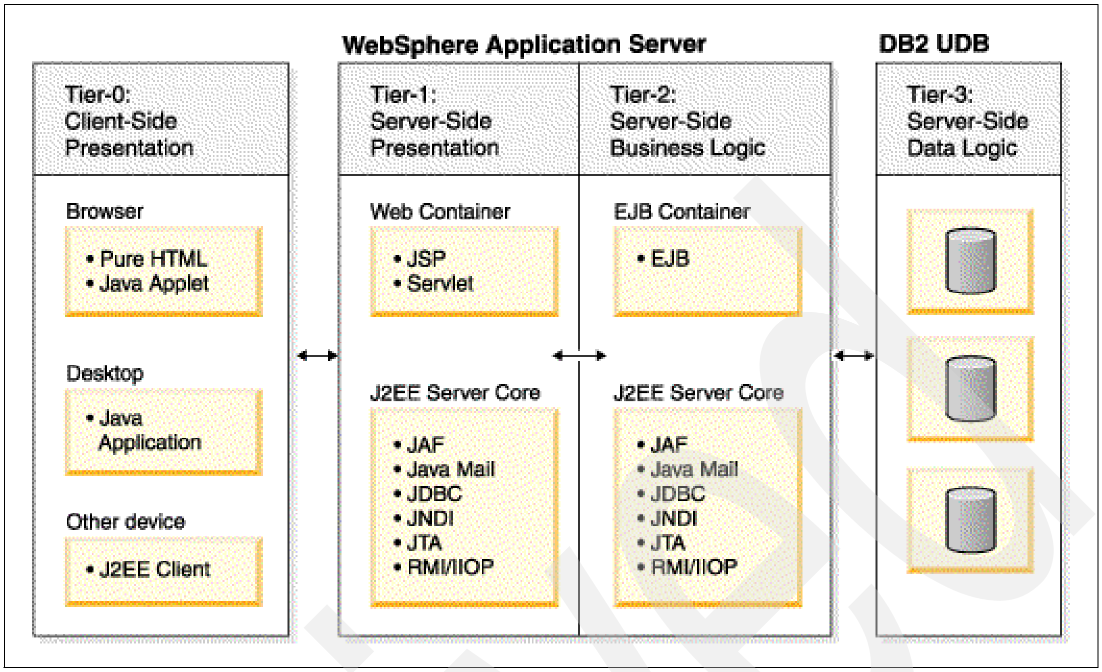

Figure 2-1 J2EE application model

Java Database Connectivity (JDBC)
JDBC defines the standard Application Programming Interface (API) for accessing relational database systems, such as DB2, from Java. This API is the fundamental building block for writing DB2 Java applications, and will be discussed extensively throughout this publication.

Java Naming and Directory Interface (JNDI)
JNDI provides standardized Java-based naming and directory services interfaces for accessing various types of naming providers, such as Lightweight Directory Access Protocol (LDAP), Microsoft Active Directory or Domain Name Service (DNS).

JNDI is significant to JDBC users as it is used to look up DataSource objects that can be used to connect to a database. See “Connecting using the DataSource API” on page 39 for more details on the use of DataSource objects.

Enterprise JavaBeans (EJBs)
The EJB model provides a standard for the development, deployment and execution of server-based Java business components. The EJB server itself manages the services required by the bean, such as security, transaction control and object pooling. This allows developers to concentrate on business logic rather than on the technical intricacies of database access, authentication and authorization, and so on.

EJBs are discussed in more detail in “Enterprise Java Beans” on page 19.

Servlets and JavaServer Pages
Servlets are a form of Java programs that plug into Web server environments, to dynamically generate Web content. This model defines how Java Servlets are written, and how they are to be managed on the server. J2EE extends the Servlet model through the introduction of JavaServer Pages, which make the creation of Java Servlets easier by giving the code an HTML tag language look and feel.

A more complete explanation of Servlets can be found in “Java Servlets” on page 17, while JavaServer Pages are discussed in “JavaServer Pages” on page 18. Hands-on examples on how to implement and deploy Servlets and JavaServer Pages are presented in Chapter 15, “Using Servlets to access DB2” on page 249, and Chapter 16, “JavaServer Pages” on page 279, respectively.

Other J2EE technologies

J2EE provides other enterprise technologies that are not directly related to the development of DB2 Java applications. These are:

- Remote Method Invocation (RMI) is a standard for distributed access to Java objects across the network. As a transport protocol, RMI can use the Java Remote Method Protocol (JRMP), or the Internet Inter-Orb Protocol (IIOP), which also supports CORBA distributed method calls.
- Java Authorization and Authentication Service (JAAS) provides for enterprise-wide security.
- Java Transaction API (JTA) is the interface for local and distributed transactions.
- Java Messaging Services (JMS) provides an interface to Message Oriented Middleware servers, such as IBM WebSphere MQ, for providing asynchronous messaging.
- JavaMail and JavaBeans Activation Framework (JAF) is an API through which Java applications can send e-mail.
- JavaIDL is an API to allow J2EE components to use external CORBA objects using the IIOP protocol.
- J2EE Connector Architecture (J2EECA) is an architecture for connecting transactional J2EE to existing enterprise systems.

2.3 Java 2 features

The foundation for all Java technologies is the Java 2 specification. This specification defines the language elements of Java 2 along with how the language is processed and executed. It covers everything from language syntax through to how the JVM is to operate. The following topics cover the significant features of this specification.

2.3.1 Object-oriented programming

Java was built from the ground up as a fully object-oriented language. This means that it is based around the concept of classes, which, when implemented, define objects. In fact, whenever you develop a program in Java, you are actually creating a class that contains instance variables and methods. Instance variables define the state of an object, and methods define the behavior of that object.

The benefit of truly object-oriented design is that classes can be developed once, and then be shared and reused across the applications that need the same functionality. In addition to this is the concept of class inheritance, where a base class is used to define the base states and behaviors, which are common across all implementations of the class. Another class can be developed that *extends* the base class, adding new behaviors or states to the base class for specific implementations of the class. In addition to this, inheritance allows for the base classes characteristics and states to be overridden, to allow the inheriting class full flexibility in its implementation.

In implementing the object-oriented model, Java is based heavily on C++. However, it removed some of the overly complex elements that were contained in C++. For example, Java does not allow multiple inheritances of classes. Also, whereas in C++ a programmer has to keep track of all allocated memory taken by objects, and has to explicitly free this memory when the object is no longer used, Java itself manages the object creation and the actual removal when they are no longer referenced. This process is called *garbage collection*.

2.3.2 Primitive data types

Java has a set of base or primitive data types. These are `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` and `double`. These types are machine independent and have a specific size. They are called primitive types because they do not contain any other types, in contrast to class types, for example.

2.3.3 Garbage collection

When an object is created in an object-oriented language, memory must be allocated in order to store its instance variables. In C++, there are two types of memory allocation: Stack-based and heap-based memory allocation. An object allocated on the stack is automatically deleted when it goes “out of scope”, that is, at the end of the program section (the *block*) where it had been created. Objects that are to remain in memory for a longer period must be allocated using heap storage.

With heap-based storage allocation in C++, it is up to the programmer to keep a eye on the object, and to explicitly delete the object when he is finished with it. Throughout the life of an object, the programmer needs to maintain a pointer to the object, and use this to ultimately delete the object. With a normal program this means the developer needs to maintain a list of active objects, and explicitly remove objects when they are no longer used. This leads to two problems. The first one is that coders often do not delete objects when they are no longer needed, meaning that programs grow unnecessarily. This situation is called a *memory leak*. A somewhat worse problem that often arises in C++ programs is when a program tries to delete an object that was already deleted, often resulting in a crash.

Java avoids this high maintenance issue by managing the objects that have been created. Java objects cannot be explicitly deleted as in C++, and they will be retained as long as they are referenced within a program. This could lead to a massive blow out in memory usage. Therefore Java periodically sets off a background task that goes through the current list of objects to verify whether they are still referenced or not. If they are not, then those objects, and the memory allocated by them, is freed. This process is called *garbage collection*.

2.3.4 Removal of pointers

Another area of complexity that exists in older programming languages is the concept of a pointer. A pointer contains the physical address of an area in memory. It can then be used to access that memory location. In C++, when an object is allocated on the heap, a pointer is used to maintain a reference to that object, and to ultimately free it.

Additionally, you can use pointers to move through memory using pointer arithmetic and then to use pointer variables to point to a specific data type within memory.

Although this is a powerful construct, it is an area of programming that has to be highly managed to ensure that you are actually pointing to the area you really want to, and that the area actually contains the correct type. If you get this wrong, you can cause runtime errors and ultimately program crashes. Adding to this complexity is that platforms have what is called *memory models*. This defines how memory is allocated and used within that platform, stating, for example, that all variables must start on word boundaries while other memory models just use the next available space. This makes the uses of pointers not very portable across different platforms.

As Java manages the memory requirements of the objects it creates, the need for pointers to manage objects is removed. Beyond this, the designers of Java decided to remove the use of pointers altogether, to remove the complexity of their use, and to increase the overall stability of the language. Also, the garbage collection processing going on behind the scenes would make the use of pointers very dangerous, as it moves objects around within memory to increase memory access performance.

2.3.5 No more GOTOs

One of the first things you are taught when learning a programming language is a statement called GOTO, which allows you to jump from one area of the program to another. The next thing they teach you is *do not use it*, as it makes debugging and error tracing difficult. Even though it was considered bad, most newer languages include a type of GOTO statement, as it can make error processing easier. Once an error occurs, such as a negative SQLCODE being returned from an SQL statement, the program jumps to the standard error routine to report the error and exits the program.

Java has no GOTO statement. It removes the need for this statement with tight mechanisms for error trapping and handling, using *exceptions* and the `try-catch-finally` construct. This provides a sophisticated error trapping and handling processing from within the programming language itself.

The general syntax of the try-catch-finally construct looks like this:

```
Connection conn = null;
try {
  Class.forName("");
  conn = DriverManager.getConnection(url, username, password);
  // Call a method which may throw an exception
} catch (ClassNotFoundException e) {
...
catch (SQLException e) {
} finally {
}
```

2.3.6 Java Virtual Machine

Java has a philosophy of *compile once, run anywhere*. As the language had the need to be able to run on all types of processors and operating systems, a major requirement is that code that is written and compiled on one platform can be moved to and executed on any other machine. To make this happen Java has the concept of first compiling the code in into *bytecode* that is portable, and then running that bytecode on a platform-specific *Java Virtual Machine* (JVM), as depicted in Figure 2-2 on page 15. Although this was not the first time a *software microprocessor* was thought of to allow portability, the use of one with major language initiatives made the concept more accessible and acceptable.

The concept of *compile once, run anywhere* had its biggest impact with the growth of the Internet browser and the World Wide Web. Programs that had been hardware- and software-specific would now have to run on any platform that would support a browser. Internet developers would have to develop applications to run on Windows, Unix, and Macintosh platforms seamlessly. When writing C++ applications, programs have to be written (or at least compiled) to target the individual platform. With Java, on the other hand, a single version of a program can be written to run on all the required platforms without change, by compiling it into platform-independent bytecode. This bytecode can be moved to any platform that has a JVM and run there. This shortens the development cycle for Internet-based applications. It was this that sold the technical world on Java, more than any other issue.

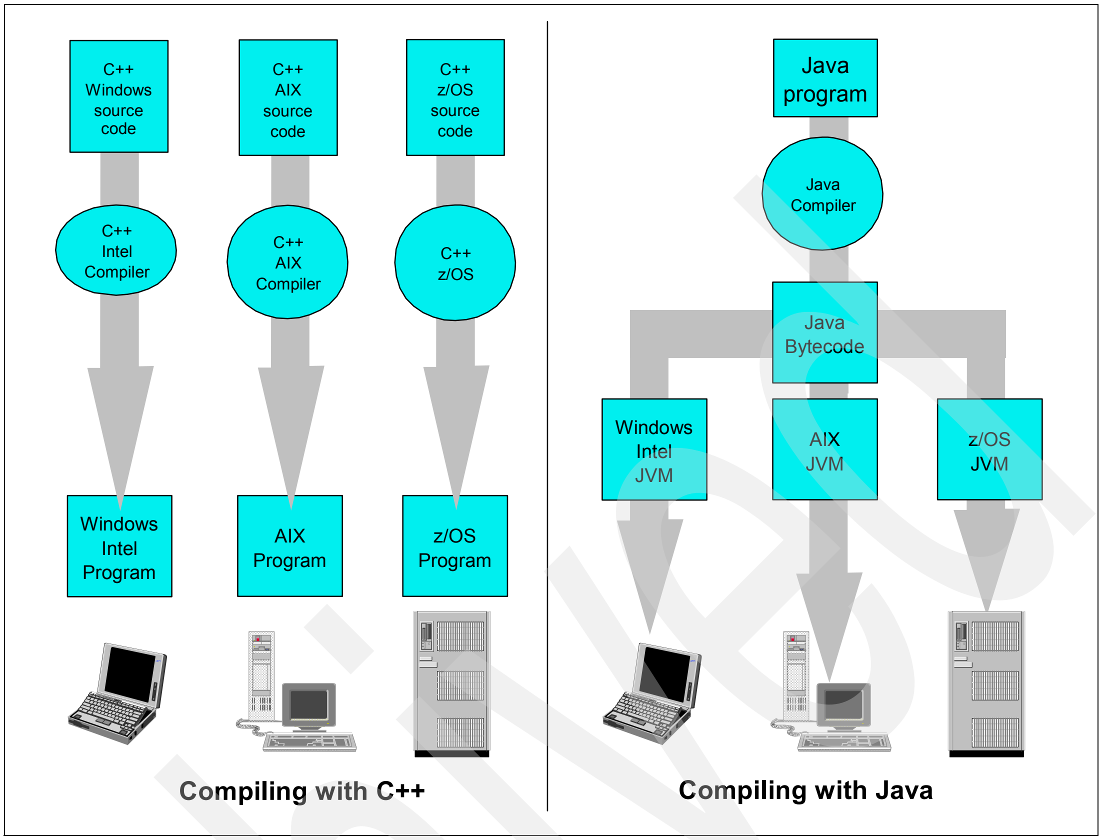

Figure 2-2 Java - Compile once, run anywhere

Much effort has been spent on improving the efficiency of the JVM. The first major improvement was the introduction of the Just in Time compiler (JIT). Originally the JVM was just a microcode interpreter. It read the bytecode, instruction by instruction, translating it into machine code. If a piece of bytecode was executed a number of times, the translation would occur every time. With the introduction of the JIT compiler, the JVM monitors how often a certain piece of code is executed; when that count exceeds a threshold, that code is translated into efficient machine code, which from now on replaces the bytecode execution of the code.

On a single user platform, this machine code is discarded when the application stops and the JVM terminates. On server platforms, however, the JVM can be reused, so when common classes are reused often and compiled, the resulting machine code can be reused by multiple users, across different applications, and performance increases significantly.

The efficiency of the JVM was further improved by optimizing the bytecode, and by improving memory management and garbage collection processing. With these improvements, Java on the single user platforms is getting close to the performance of platform-specific programs.

2.4 Java application environments

Java has evolved from a simple single device programming language, to a fully blown enterprise development system. This evolution has lead to a number of distinct programming solutions, from a simple stand-alone program, through a server-based Web page generation component, and ending up with a business process object that encapsulates and manages business logic across the enterprise.

2.4.1 Stand-alone Java applications

By far the simplest form of a Java program is the stand-alone Java application. These are complete programs that manage all the functionality within one unit of code, although they can make use of external classes. They also tend to be console applications, which means that they receive input and send output in text format from the command line. In z/OS terms this means that the program is executed directly in the Unix System Services environment. As this type of program is similar to other mainframe batch programs, a stand-alone Java application can be run in batch on a z/OS system, using the BPXBATCH utility (see “Running the Hello application from MVS batch” on page 147 for an example).

A stand-alone Java program application has a `main()` method, which is the entry point for the program. It can accept command line arguments that are placed in the `String` parameter `args[]`.

Once the application has been coded, it is compiled into byte code by running the **`javac`** command. The output of this process is a class file. The program is executed by running the **`java`** command.

2.4.2 Java applets

These are Java programs that can be included in an HTML page using the `<applet>` tag. They are downloaded to the client machine by the browser that is displaying the host HTML page, and executed under the browser’s JVM. Applets are normally small programs that extend the functionality of the browser. Also, as they are browser based, they tend to be graphical in nature, although they can be used to access and execute all features of a browser, such as playing audio files and processing user input.

As applets are client-based applications, they are restricted in what they can do to the client machine executing them. The biggest restriction is that an applet cannot access the client’s hard drives to either read or write data. This is to protect the client from intrusions from applications (which they may or may not know) that are loaded from the Web. Java does have a method to allow applets to perform disk access. It is called *applet signing*. This verifies that the applet comes from a certified source, and it is up to the user to decide whether he trusts that source enough to give the applet access to his or her computer’s resources.

Applets are an extension of the `Applet` class, and use different methods to allow them to react to the states of the host browser page. The four main methods of the applet are shown in Table 2-1.

Table 2-1 Methods of a Java applet

Method	Operation
`init()`	Performs the initialization of the applet when it is first executed. This is used to set the layout of the applet and is always overridden.
`start()`	Called whenever the applet on the browser comes into view. Also called after the `init()` is executed. This method can be overridden if you wish to start running expensive processes that only need to run when the applet is visible, such as graphic processes.
`stop()`	Executed whenever the applet goes out of view. This method will be overridden to stop processes that are not required while the applet is hidden.

Method	Operation
`destroy()`	Called just before the applet is unloaded, and allows final release of the applet resources

Applets are downloaded from the Web server onto the clients each time they are executed. A further download is started for each class the applet requests. This means there can be a performance hit if the applet is large or requests to many classes. Therefore applets tend to be typically small single-function processes.

2.4.3 Java Servlets

A Java *Servlet* is a piece of Java code that is written to operate within the confines of a Java-enabled server. They extend the functionality of Web application servers, and provide a component-based platform-independent method for building Web applications. Although Java allows for other protocols to access a Servlet, most of the requests emanate from a Web browser and use the HTTP protocol.

The Servlet is mapped to a URL. On receiving a request for that URL, the server executes the Servlet by invoking the service method of the Servlet. The Servlet then processes the request and responds to it. Normally the response is in the form of an HTML document to produce dynamic Web content.

Servlets are run in a section of the Web server called the *Servlet container*. The container manages the running of the Servlet instances, and provides the threads when a request is made to execute a Servlet.

The beauty of this architecture is that all the business logic is coded and executed on the server itself. All that is required to exist on the client machine is a browser, and the mechanism to communicate with the server.

Another feature of this architecture is that all connections to resources required to process the Servlet only need to be present and available on the server platform itself. If a process requires access to back-end databases, this is managed and executed from the server, not the client, so all database access software is present in one place and does not need to be replicated to each client.

To code a Servlet, you extend one of two Servlet classes, either the `GenericServlet`, which is used to define a protocol-independent Servlet, and the more often used `HttpServlet` class, which uses the HTTP protocol for use on the Web. The significant methods of an HTTP Servlet are shown in Table 2-2.

Table 2-2 Methods of a HTTP Servlet

Method	Operation
`init()`	Performs the initialization of the Servlet when is it started. This can occur when a Web server is started or on the first request for the Servlet, if it has not already been started.
`service(HttpServletRequest req, HttpServletResponse resp)`	This method processes the request sent to the Servlet. Usually it returns the output of this process to the client in the form of dynamic Web pages by opening a `PrintWriter` and printing out HTML tags and data.

Method	Operation
`destroy()`	Called by the Servlet container to indicate that the Servlet instance is to be terminated. It will only be called when all threads using the Servlet have terminated. Its purpose is to allow resources held by the Servlet to be freed.

Although Servlets provide efficient performance with a simple architecture, there can be problems due the fact they are Java programs that implement dynamic Web pages using hardcoded HTML. This leads to three issues:

- As the HTML is hardcoded, it can be inconvenient to implement changes to the output Web pages, as the program needs to be recompiled every time a page is changed.
- Second, it is difficult to support different languages in the Web output by determining the user's language and location and then displaying them in the dynamic page. A way around this is to build Servlets for specific languages and regions, but this leads to replication of the HTML and further complicates the change process.
- The third issue is caused by the issue of responsibilities. The rise of the Web has given rise to the Web designer, normally a person with a graphics and user interface design background, and usually not grounded in the development of Java programs. By mixing straight Java code and HTML, the Servlet model makes it difficult to separate the responsibilities of the Web designer and the Java developer in the development process. This in turn complicates the whole development and maintenance process of Servlets.

2.4.4 JavaServer Pages

JavaServer Pages (JSPs) are a natural extension of the Java Servlet. They came about to make the separation of responsibilities between the Web designer and Java developer more distinct. They also allow for a more dynamic method of change for dynamically generated Web pages.

JSPs achieve this by reversing the concepts of the Servlet. In the Servlet hardcoded HTML is slotted in between the logic of a Java program. A JSP is a text document made up of static HTML and XML like tags, and scriptlets that encapsulate the logic that generates the dynamic content.

One of the significant features of a JSP is that it is translated into Java Servlet code, which is then compiled using the standard Java compiler before it is executed in the JVM. The JSP is stored in a section of the Web server called the JSP container. This container manages the translation and compilation of the page into the Servlet. It checks whether a translation of the page is required, due to the page being updated since the last execution. If it has not been changed, the Servlet is called directly.

Although the JSP came about to rebalance the development of Java Servlets towards the Web designer, by making it an HTML page surrounding the Java program logic, the problem that the two distinct technologies are mixed remains. Although it made it quicker to develop a dynamic Web page, the heavy mixing of HTML and programming logic created a maintenance nightmare, similar to the days of pure Java Servlets.

As an alternative, an architecture called Model 2 design was put forward. This basically means that a JSP contains only the graphical Web elements within the page. In this model, the page is only responsible for the *view*, which is how the data is presented to the browser. Almost all programming logic is pushed back into pure Java programs, which contain the business logic and interact with the JSPs to present the user interface.

To extend the Model 2 design architecture, the JSP specification now allows the creation of a Tag Library. This library is used to store custom tags that are defined to the application and define a call to the Java code to perform the business logic. All the Web developer needs to know is the tag and what function is to be run, while the Java developer only codes the business logic for the function, without writing presentation layer code. Thus the separation of functions between Web developer and Java programmer are maintained, and Web pages can be changed without pouring through Java code.

2.4.5 JavaScript

Despite the name, JavaScript and Java have very little in common. JavaScript is purely a scripting language for Web pages developed by Netscape to simplify dynamic Web site development.

As JavaScript operates as an extension of HTML, it is completely executed on the client under the control of the browser. JavaScript code is interpreted by the browser and is not compiled or executed under the JVM.

2.4.6 Java Beans

The Java Beans API defines a software component model for Java to support the development of applications using visual builder tools.

The most obvious use of Java Beans is the development of GUI applications, where you use a tools palette to drag and drop visual components—such as buttons, text entry fields, and list boxes—onto a free-form surface. You then set up the component's properties (such as a button's text), and connections between the components (for example, to enable or disable a button depending on whether or not the user entered some text in a text entry field).

2.4.7 Enterprise Java Beans

Despite the similar name, Enterprise Java Beans (EJBs) have a very different focus compared to Java Beans. While Java Beans are essentially client-side components, EJBs set a platform-independent standard for the development of server-side business logic components. Using a set of classes that conform to the EJB standard, a developer can create, assemble, and deploy component objects that implement the business logic of the enterprise. Once deployed, the component is managed by the *component transaction monitor* (CTM), which looks after the infrastructure issues such as instance swapping, resource pooling, and activation.

The first EJB specification 1.1 appeared in J2EE 1.2, while J2EE 1.3 extended this specification with the introduction of EJB 2.0, introducing performance enhancements and the message-driven EJB. The most recent release, EJB 2.1, was released with J2EE 1.4, which is the most current specification at the time this publication was written.

Why use EJBs

One reason for the development of EJBs was to separate middleware-oriented code, responsible for persistence, security, transaction control, and so on, from the business logic. Beans encapsulate the business logic mainly within the session bean, with data access controlled using the entity beans. Issues such as security, communication, and actual bean location are managed by the container in which the beans are run, and once set are no concern of the client programs that are ultimately going to call these EJBs. This also increases the ability to scale the applications, as beans can be spread across servers.

The three flavors of EJBs

There are three distinct types of EJBs:

- Session beans

  There are two types of session beans:

  – Stateless session beans (SLSBs)

  – Stateful session beans (SFSBs)

- Entity beans

  There are two types of these beans:

  – Bean managed persistence (BMP)

  – Container managed persistence (CMP)

- Message-driven beans

Each of these bean types handles different elements of a business application. *Session beans* control business logic between the client and the application, while *entity beans* represent business objects, typically being an in-memory representation of a row in a database table. Finally, *message-driven beans* provide a link between asynchronous messaging and the synchronous session and entity beans.

Stateless session beans (SLSBs)

Stateless session beans are the most simple implementation of the EJB model. These beans are memory-based components that do not maintain data between calls. If data is to be processed by an SLSB, it has to be passed into the bean with each invocation of any of the bean's methods.

As stateless session beans match the stateless model of HTTP, they are popular for handling access from Web applications.

Because SLSBs do not maintain data between calls, they are very generic in nature, which means that they can easily be pooled for efficient reuse. As the method call is the same to all clients, and no data is stored for a call, clients can use any bean (of the same type of bean that they require) from the pool. Once a SLSB has finished processing a client's request, it can be returned to the pool for reuse.

Stateful session beans (SFSBs)

The use of SLSBs is somewhat cumbersome for applications that require data to be maintained across invocations. SLSBs can be used for such applications by pushing the session data back to the client, which must remember the data between calls and pass it to the bean with each method call. Another solution is to use *stateful session beans*.

Stateful session beans maintain data for the whole conversation with the client and across client requests. A good example of such a bean is a shopping cart EJB, which remembers all the products that a person has requested in their current session, and maintains information such as the total cost of all the items purchased.

Entity beans

Entity beans are beans that maintain persistent data, that is, typically data stored in a database. Although it is not necessarily the case, normally relational databases are the target for entity beans.

The entity controls persistence by maintaining both a copy of data as a stateful component in memory, and a matching entry in persistent storage.

There are two types of entity EJBs: Bean Managed Persistence (BMP) entity beans and Container Managed Persistence (CMP) entity beans.

Bean managed persistence (BMP)

In the entity EJB with bean-managed persistence, the bean itself implements the code for managing the connection between the in-memory representation of the bean and the persistent representation in the database. The coder of BMP beans has a fixed set of methods that he or she fills with the code that will physically process the database row. Some of these methods are:

- ejbRemove() contains the code to physically remove the data from the persistent store. This method normally contains SQL DELETE statements.
- ejbCreate() is called when a new instance of an EJB class has to be created. It usually executes SQL INSERT statements and returns the primary key values.
- ejbLoad() initializes the bean instance with the values from the database. It normally contains SQL SELECT statements using the primary key value.
- ejbStore() updates the database with the current values from the in-memory representation, that is, it writes back any changes to that in-memory representation. This is typically done with SQL UPDATE statements.

Some of the other methods allow processing to handle activation and deactivation of the bean, as well as methods to reactivate the bean to allow reconnection to the database, and a corresponding method to allow connections to be dropped.

Further setter and getter methods may be defined in the EJB to set column values in the beans data.

It is important to understand that although specific SQL is coded in the bean's methods, it is the container that decides when the methods are executed, and BMP beans must be coded to allow for this. Finally, when the container has finished with an instance of a bean, it is returned to a pool for reuse.

Container managed persistence (CMP)

BMP beans offer flexibility, but can be difficult to maintain and repetitive to produce. The alternative is to use container managed persistence (CMP) entity beans. With these beans, the definition of the physical data store is configured in the container deployment descriptors (using XML), and all access to the bean, and between the bean and the physical data, is managed using this descriptor file.

Message driven beans

These beans were added in the EJB 2.0 specification, and handle the interfacing of asynchronous JMS messages and synchronous EJB components.

3

Accessing DB2 from Java

This chapter of the publication describes how a Java application can access DB2 for or OS/390 and z/OS databases. It describes the processes and components that can be used from either a Windows development platform or running under Unix System Services on z/OS or OS/390. The topics discussed are:

- Java and JDBC
- Selecting and using JDBC drivers
- The Universal Driver for SQLJ and JDBC
- Different ways to connect to a DB2 for z/OS and OS/390 system
- Using the DriverManager and DataSource APIs
- SQLJ versus JDBC

3.1 JDBC basics

When accessing relational data from a Java application you cannot go far without first understanding JDBC. All database access occurs through JDBC, from connecting to a database to actually processing the data in that database.

Java Database Connectivity (JDBC) is an application programming interface (API) that the Java programming language uses to access different forms of tabular data, as well as some hierarchical systems. The list includes full-blown relational databases such as DB2, down to spreadsheets and flat files. The JDBC specifications were developed by Sun Microsystems together with relational database providers, such as IBM and Oracle, to ensure the portability of Java applications across databases and platforms. At this time there have been three major releases of the JDBC API specification, the last one being JDBC 3.0, which was released in February 2002.

The APIs are a set of classes and related methods that support full SQL functionality. The classes fall into three categories:

- Connecting to a database, communicating the data layouts (or metadata) to the application, and then processing the data within that database
- Exception processing or error handling
- Translating the native database data types to Java classes

The API is defined in the classes and interfaces that make up the Java packages:

- java.sql.*
- javax.sql.*

An interface is a Java construct that looks like a class but does not provide an implementation of its methods. In the case of JDBC, the actual implementation of the JDBC interfaces is provided by the database vendor, as a JDBC driver.

This may sound complex, but in fact it provides portability in that all access using JDBC is through standard calls with standard parameters. Thus an application can be coded with little regard to the database you are actually using, as all the platform-dependant code is stored in the JDBC drivers.

Having said that, JDBC also has to be flexible in what functionality it does and does not provide, solely based on the fact that different database systems have different levels of functionality. For example, JDBC provides a method called `getArray()`, which supports a column type of ARRAY. This column type is not supported by DB2, and therefore such a function is meaningless. When a DBMS does not support such functions, the JDBC standard says that if the method is executed in a program, the driver has to raise an `SQLException`.

So although JDBC allows for greater portability, it is important to understand what functionality is available in the database system that you are using.

The other side of the coin is that vendors that provide JDBC drivers can add functionality htat is database system specific. These methods are known as extensions. Although they may provide benefits, they should also be used carefully, as they limit portability.

3.1.1 JDBC driver types

As mentioned before, the JDBC drivers provide the physical code that implements the objects, methods, and data types defined in the JDBC specification. The JDBC standard defines four different types of drivers. The distinction between them is based on how the driver is physically implemented, and how it communicates with the database. The driver types are numbered 1 to 4 (Figure 3-1).

> **Important:** The naming of these driver types does not represent improvements or versioning of the drivers. Type 2 drivers are not a later version of Type 1 drivers, and Type 4 is not a fuller implementation of a standard than a Type 2 driver.

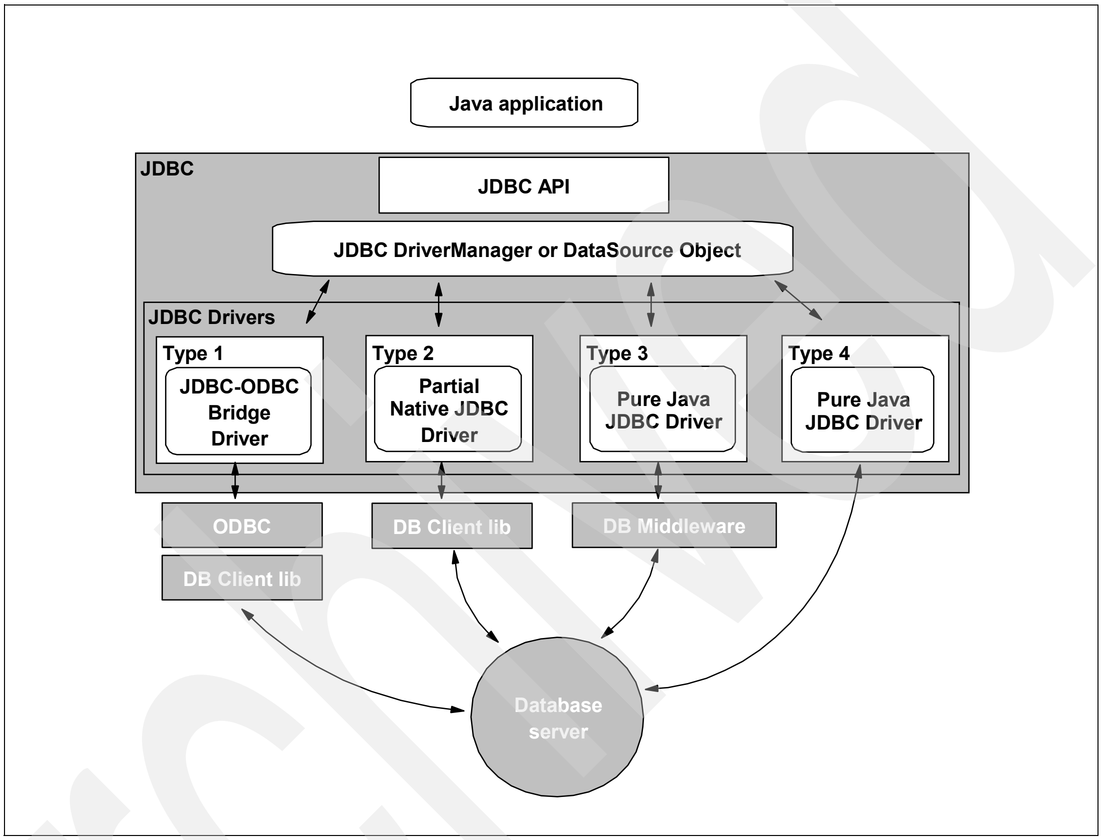

Figure 3-1 Java Database Connectivity (JDBC) API and driver types

Type 1

This is the oldest type of driver. It was provided by Sun to promote JDBC when no database-specific drivers were available. With this driver, the JDBC API calls an ODBC driver to access the database. This driver type is commonly referred to as a JDBC-ODBC bridge driver. Its shortcoming is that ODBC must be implemented and installed on all clients using this driver type. This restricts it to platforms that have ODBC support, and as such is mostly Windows centric. As it also has to translate every JDBC call into an equivalent ODBC call, the performance of a Type 1 driver is not all that great.

This type is no longer commonly used, as virtually every database vendor nowadays supports JDBC and provides their own (vendor-specific) driver. For DB2, it is not officially supported.

Type 2

Here, the JDBC API calls platform- and database-specific code to access the database. This is the most common driver type used, and offers the best performance. However, as the driver code is platform specific, a different version has to be coded (by the database vendor) for each platform.

This type of driver is provided with Version 7 of DB2 for z/OS and OS/390. It is available for Version 5 through APAR PQ19814, for Version 6 through APAR PQ36011, and it is part of the base code with DB2 Version 7 (and Version 8 when it becomes generally available).

Both Type 1 and Type 2 drivers are partially written in Java. They also require a platform-specific client library, usually written in C. To be able to run a Java application that wants access to a database through a Type 1 or Type 2 driver, you need both the driver and the platform-specific client library on the client machine.

This is in contrast to Type 3 and Type 4 drivers. They are pure Java drivers, which makes them easier to port and to deploy, as they do not require a platform-specific client library to go with the driver.

Type 3

With this driver type, the JDBC API calls are routed through a middleware product using standard network protocols such as TCP/IP. The driver itself is written in Java.

The middleware translates these calls into database-specific calls to access the target database and returns data to the calling application. In the case of DB2, this task is performed by a program called the *JDBC applet server*. DB2 for z/OS and OS/390 does not supply a Type 3 driver. DB2 for Linux, Unix and Windows still has a so-called network or net driver, but if you are not using it already, you are strongly encouraged to use the Type 4 driver instead.

Type 4

A Type 4 driver is fully written in Java, and accesses the target database directly using the protocol of the database itself. In the case of DB2, this is DRDA. As the driver is fully written in Java, it can be ported to any platform that supports that DBMS protocol without change, thus allowing applications to also use it across platforms without change.

This driver type has been implemented through the IBM DB2 Universal Driver for SQLJ and JDBC, which is first provided with Version 8 of DB2 UDB for Linux, Unix and Windows. The driver will also be made available for DB2 for z/OS and OS/390 Version 7 with the PTF for APAR PQ80841. This PTF was not available at the time this Redbook was published. This driver is also known as the DB2 Universal Driver for Java Common Connectivity or JCC. In addition, the Universal Driver also provides Type 2 connectivity.

For more information about the Universal Driver, see “The IBM DB2 Universal Driver for SQLJ and JDBC” on page 27 and Chapter 12, “The DB2 Universal Driver” on page 221.

JDBC drivers and driver types for use with DB2 z/OS and OS/390

There are a number of different JDBC drivers available for DB2 on the z/OS and OS/390 platform, and some great new ones just around the corner, each having different characteristics and levels of cross platform compatibility. Table 3-1 gives an overview.

Table 3-1 JDBC drivers and driver types for use with DB2 z/OS and OS/390

Jar file(s)	Driver name	Driver type	JDBC level	Legacy driver	Usage notes
db2sqljclasses.zip	COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver	2	1.2	Y	Can only be used on z/OS or OS/390 platform. Contains all classes necessary for preparing and running JDBC and SQLJ programs.

Jar file(s)	Driver name	Driver type	JDBC level	Legacy driver	Usage notes
db2sqljruntime.zip	COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver	2	1.2	Y	Can only be used on z/OS or OS/390 platform. Contains only execution classes for JDBC and SQLJ programs. Cannot be used to execute the SQLJ preprocessor (translator).
db2j2classes.zip	COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver	2	2.0	Once JCC becomes available on z/OS	Can only be used on z/OS or OS/390 platform. JDBC extension file jdbc2_0-stdext.jar does not have to be installed as JDBC 2.0 extensions are included in the class file.
db2java.zip	COM.ibm.db2.jdbc.app.DB2Driver	2	2.0	Y	This is the 'old' DB2 for LUW Type 2 driver, also known as the 'app. driver'. This driver is only mentioned here as it can be used with DB2 Connect™ running on the client machine, to access data on z/OS or OS/390 DB2 databases. You are kindly encouraged to start using the new JCC Type 2 driver.
db2java.zip	COM.ibm.db2.jdbc.net.DB2Driver	3	2.0	Y	Can be used with DB2 for Linux, Unix and Windows. This is the 'old' 'net driver'. You should look at replacing this driver with the Type 4 flavor of the new Universal Driver.
db2jcc.jar	com.ibm.db2.jcc.DB2Driver	4	3.0	N	Can be used on Linux, Unix and Windows platform, as well as z/OS or OS/390 platforms. Pure Java code driver that uses TCP/IP across DRDA to directly connect to database. Does not require DB2 Connect code to be present on client machine to access remote databases.
db2jcc.jar	com.ibm.db2.jcc.DB2Driver	2	3.0	N	Available on Linux, Unix and Windows, and soon on z/OS and OS/390.

3.1.2 The IBM DB2 Universal Driver for SQLJ and JDBC

As mentioned before, IBM has released a brand-new JDBC driver with DB2 UDB Version 8 for Linux, Unix and Windows called the *IBM DB2 Universal Driver for SQLJ and JDBC* (also known as the DB2 Universal Driver, or the DB2 Universal Driver for Java Common Connectivity, or JCC).

It supports Type 2 and Type 4 connectivity to DB2 Version 8 for Windows, AIX 32-bit and 64-bit, Solaris 32-bit and 64-bit, Linux IA-32 and z31-bit and HP 32-bit and 64-bit. The Type 4 driver has been available since V8 went generally available. The Type 2 driver support was added with FixPak 2 (V8.1.2). Using the Type 4 driver you cannot only access DB2 for LUW systems. The Universal Driver (DB2 for LUW V8 Fixpak 3), at the time of writing of this publication, also supports:

- Type 4 connectivity to DB2 for iSeries™ V5 Release 1, PTF SF99501, SI08452
- Type 4 connectivity to DB2 for iSeries V5 Release 2, PTF SF99502, SI08451, SI08479, SI08478
- Type 4 connectivity to DB2 for zOS V6, PTF UQ72081, UQ72082
- Type 4 connectivity to DB2 for zOS V7, PTF UQ72083
- Type 4 connectivity to Cloudscape™ Network Server 5.1 FixPak 1 and later

Ultimately, the Universal Driver will also be released with Type 2 and Type 4 connectivity for DB2 for z/OS and OS/390. Although not officially released for DB2 for z/OS and OS/390 at the time of writing of this publication, we used the Type 4 driver for many of the examples in this publication, for example, when running a Java program in Unix System Services connecting through the Type 4 driver to DB2 for z/OS and OS/390 (via DDF). As a Type 4 driver is a pure Java driver, it is sufficient to FTP the required jar files from the Windows platform (where the driver is available with DB2 for Linux, Unix and Windows Version 8) to the OS/390 platform. You only need to adjust the CLASSPATH, and you are up and running. We had no problems whatsoever with the functionality of the drivers on both platforms, a nice illustration of its portability, and that of the programs using it. (Note, however, that you must have the proper licensing agreements in place before you do this.)

Attention: As mentioned above, we copied the class files shipped with DB2 for LUW, because the official z/OS Type 4 Universal Driver was not available when this book was written. This technique, although it worked fine for our examples, is NOT officially supported. You should NOT attempt to use the DB2 Universal JDBC Driver shipped with DB2 for Linux, Unix and Windows on a z/OS platform. Although it may work, as it did for us, it is not supported and may lead to unpredictable results

The new driver has been written from scratch and is not based on any of the existing JDBC/CLI drivers. The Universal Driver is architected as an abstract JDBC processor that is independent of driver-type connectivity or target platform. The IBM DB2 JDBC Universal Driver is an architecture-neutral JDBC driver for distributed and local DB2 access.

Since the Universal Driver has a unique architecture as an abstract JDBC state machine, it does not fall into the conventional driver-type categories as defined by Sun. Because the Universal Driver is an abstract machine, driver types become connectivity types.

This abstract JDBC machine architecture (see Figure 3-2 on page 29) is independent of any particular JDBC driver-type connectivity or target platform, allowing for both all-Java connectivity (Type 4) or JNI-based connectivity (Type 2) in a single driver. A single Universal Driver instance is loaded by the driver manager for both Type 4 and Type 2 implementations. Type 2 and 4 connections may be made (simultaneously if desired) using this single driver instance.

Programs select the desired type of connectivity by using different URL syntax when making the connection (see “Establishing a connection” on page 38 for details on how to code the different URLs).

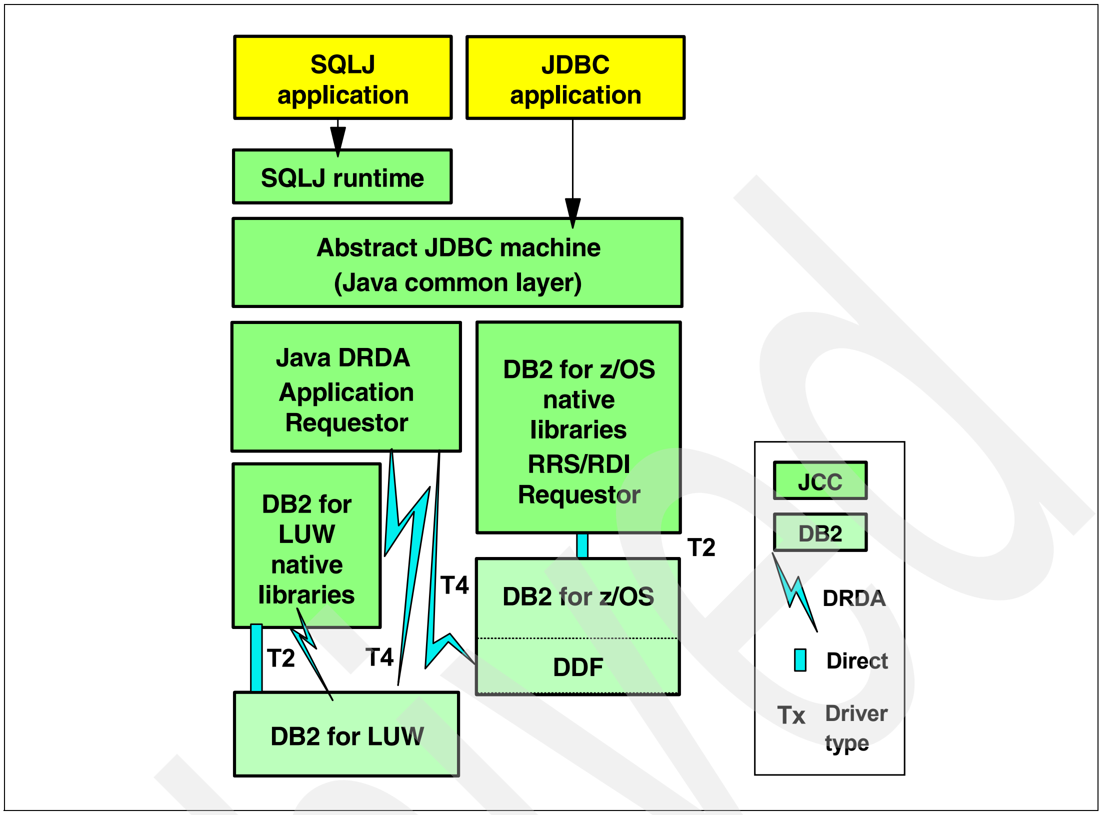

Figure 3-2 Universal Driver architecture

Type 4 connectivity

The Type 4 driver is based on an open distributed database protocol, known as Distributed Relational Database Architecture (DRDA). This driver connects to (remote and local) subsystems using DRDA over TCP/IP. This eliminates the need for DB2 Connect, but all servers must be configured for TCP/IP communication. (However, for licensing purposes, you still need a DB2 Connect to be able to use Type 4 connectivity to a DB2 for Os/390 and z/OS.)

Type 2 connectivity

For the Type 2 driver, the story is a little bit more complicated.

Connecting to a DB2 for Linux, Unix and Windows

When the application (for example, WAS on Windows) is running *on the same machine* as the DB2 UDB database resides, the Type 2 driver uses a shared memory connection.

When the application is running *on a different machine*, the Type 2 driver uses a DRDA connection to get to the database.

Connecting to a DB2 for z/OS

The use of a Type 2 driver on z/OS or OS/390 assumes that the application runs on the same LPAR as the DB2. It uses the RRS Attachment Facility (RRSAF) when connecting to z/OS or OS/390-based DB2 subsystems.

The use of the new Universal Driver also greatly simplifies the development and deployment of SQLJ applications (see “Preparing SQLJ programs to use static SQL through WSAD” on page 157 for details), and offers several useful (but proprietary) extensions, which are covered in Chapter 12, “The DB2 Universal Driver” on page 221.

3.2 Different ways to connect to a DB2 for z/OS and OS/390

In this section we show some of the commonly used ways to connect a Java application to a DB2 for z/OS and OS/390 subsystem. For this discussion, the type of Java application is not important. Therefore, when referencing a Java program, that can be a Java application, an applet, a Servlet, or an EJB.

3.2.1 Direct (T2) connection to a local DB2 subsystem

This configuration is illustrated in Figure 3-3. In this case, the Java application runs under Unix System Services (USS), and talks to DB2 through a Type 2 driver. A JDBC call is translated by the Type 2 driver into SQL statements, and executed through the RRS attachment facility by DB2.

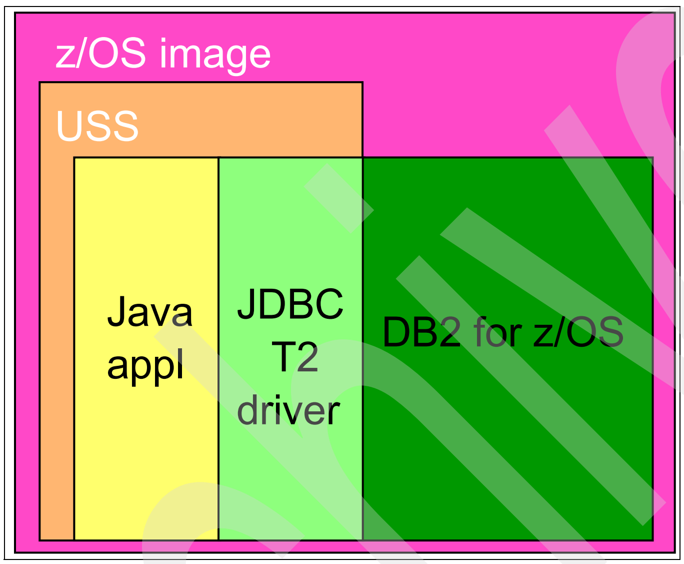

Figure 3-3 Local T2 connection

As the application is local to the DBMS, and we use a Type 2 driver that can do 'native' calls to DB2; this configuration normally provides the best performance. You also use this configuration when running your WebSphere Application Server (WAS) on z/OS. Your Java application can run under the control of WAS, and talks to DB2 through the Type 2 JDBC driver, and a local RRS attachment.

3.2.2 Using the Type 4 driver to talk to a local DB2 for z/OS and OS/390

In this configuration (Figure 3-4 on page 31) we use a Type 4 driver to connect to a local DB2 for z/OS and OS/390 subsystem through DRDA and DDF. Note that the application and the DB2 system are running on the same z/OS image (LPAR).

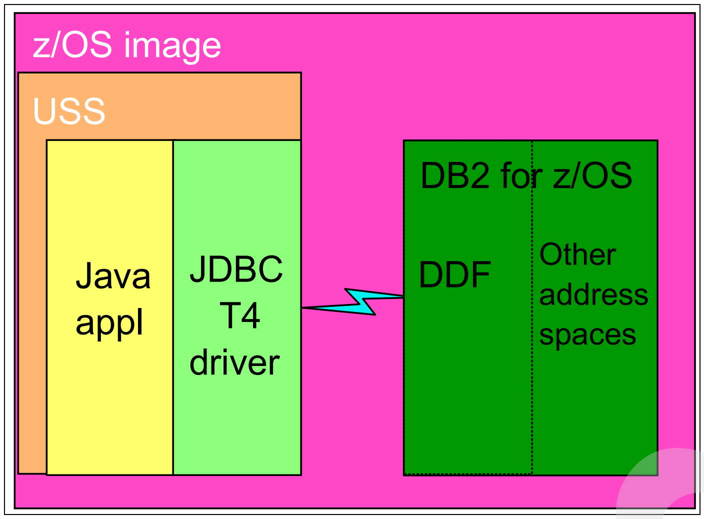

Figure 3-4 Local application using Type 4 driver

This is not a recommended setup for a production environment, as all SQL requests go through DRDA over TCP/IP (so through the TCP/IP stack) into DB2's DDF address space. Even though DRDA is a very efficient database communication protocol, it is normally more expensive than going through the local RRS attachment that we discussed in the previous section.

The main reason why this configuration is shown here is that, at the time of writing of this publication, the new JCC Type 2 driver was not available for DB2 for z/OS, so we used the Type 4 driver during most of our samples.

For testing purposes, like our examples, this configuration is OK. As the functionality of the Universal Driver is (almost) identical for the Type 2 and Type 4 driver, you only need to change the URL to switch between both driver types and you are done.

3.2.3 Type 4 connectivity from a non-z/OS platform

Here we look at another setup using the Type 4 driver. In this case, our application is running on a non-z/OS platform (Figure 3-5 on page 32). As we are using a pure Java driver, the application can be running on any hardware and software, as long as it supports a compatible JVM. To talk to DB2 for z/OS and OS/390 from our Java application, we use a Type 4 driver. This way we can communicate directly to DB2 for z/OS and OS/390 through DRDA. All connections to DB2 for z/OS and OS/390 come into the system through DDF (since that is the DB2 address space that knows how to 'talk' DRDA). Note that the application can be an applet running in a browser.

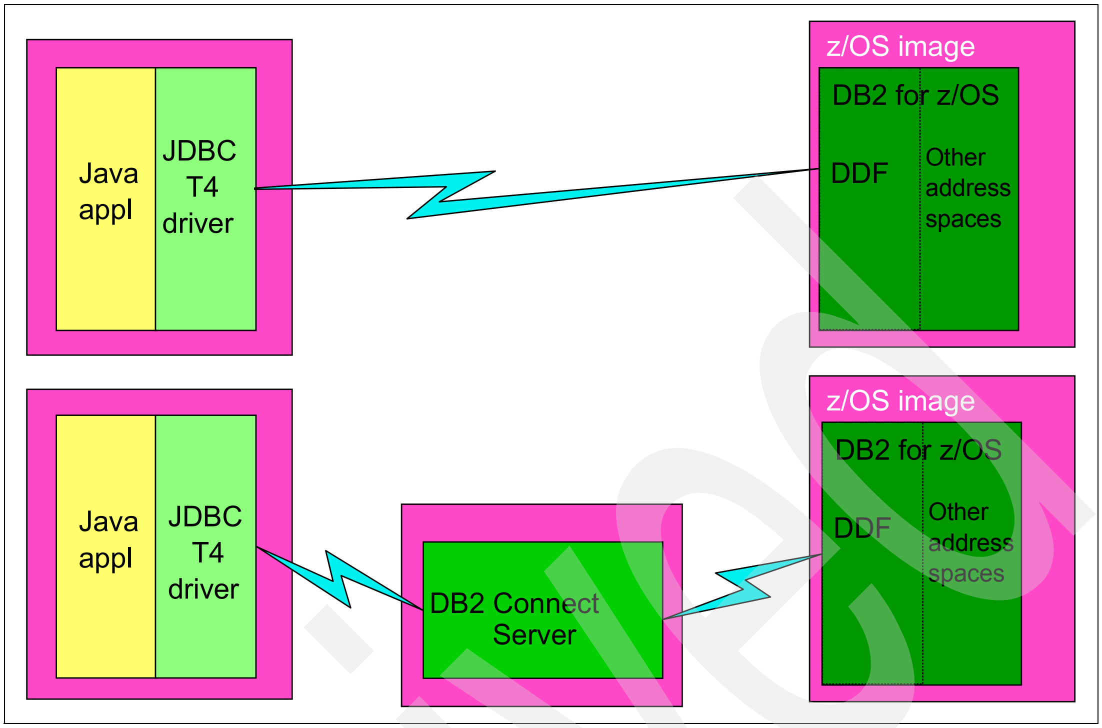

Figure 3-5 Type 4 connectivity from a non-z/OS platform

This configuration is very common when developing applications. For example, when you are using WebSphere Studio Application Developer (WSAD), running on a workstation to develop your Java applications, you can use the Type 4 driver to test your application, and run it against data on a DB2 for z/OS system.

The lower half of Figure 3-5 shows a configuration using the DB2 Connect Server. When using the Type 4 driver, you do not need DB2 Connect to be able access data on a DB2 for z/OS and OS/390 system. So you may ask yourself, why is there a DB2 Connect Server in the figure, if it is not required? The reason is that DB2 Connect Server provides functionality that is not provided by the Type 4 driver. These functions include sysplex awareness and connection concentration. These functions can be very valuable in large installations. For a more detailed description of these features, please see *Distributed Functions of DB2 for z/OS and OS/390*, SG24-6952.

Note: DB2 Connect Server is a component of multiple DB2 Connect products, namely:

- DB2 Connect Enterprise Edition (EE)
- DB2 Connect Application Server Edition (ASE)
- DB2 Connect Unlimited Edition (CUE)

3.2.4 Type 2 connectivity from a non-z/OS platform

As a Type 2 driver is normally a local driver, and in this configuration the DB2 for z/OS and OS/390 system is not local to the JDBC driver, you need something in the middle. This ‘thing in the middle’ is of course DB2 Connect. The configurations in Figure 3-6 on page 33 are often used when the WAS system is running on a different machine (and/or platform) than the DBMS (DB2). This is often done for security reasons. People want to make it as hard as possible for hackers to get to the corporate data. Therefore, in many installations it is not allowed to have the data on the same physical machine as the Web server.

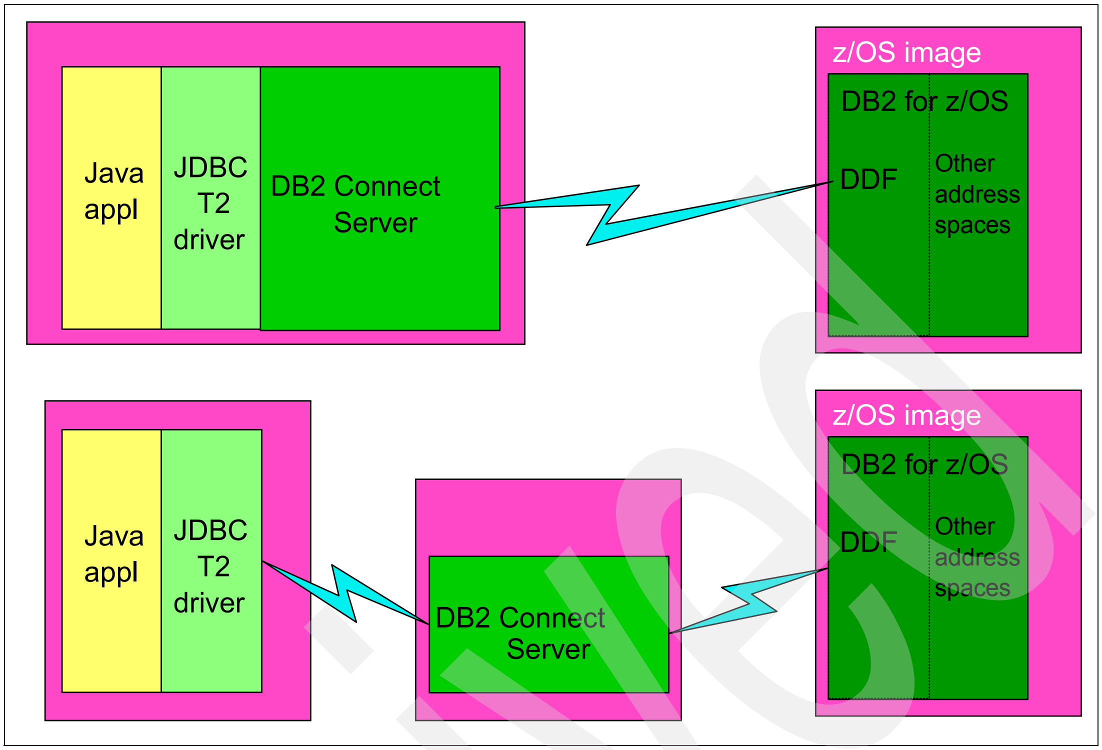

Figure 3-6 Type 2 connectivity from a non-z/OS platform

But as you can see in the figure above, this is not a problem. You run your Java applications under an application server on one machine, and use the Type 2 driver to talk to DB2 Connect Server, which passes on your database requests to DB2 for z/OS and OS/390 on another machine. The database communication protocol that is used for this is DRDA. Note that from the DB2 for z/OS and OS/390 point of view, these are remote connections coming into DDF.

The next burning question you may have is, why not use a Type 4 driver in all cases, and forget about the Type 2 driver in the scenarios where the Java application is running on another platform? In both cases DRDA involved, so performance is likely to be equivalent. The answer is that the Type 4 driver does not support all the functions that the Type 2 driver does. The most important one being that *currently* two-phase commit is not supported by the Type 4 driver. However, this is likely to change very soon.

3.2.5 DB2 for z/OS and OS/390 as a DRDA Application Requester

People that are into very high availability, running their WAS systems on zSeries® hardware in a parallel sysplex, often have the same security guidelines to follow as mentioned before: No Web server and database server on the same machine.

In this case you can have a WAS running your Java applications that talks to a Type 2 JDBC driver, that talks through RRS to a DB2 system (DB2A) that is local to the machine running the WAS. This local DB2 then ‘routes’ all requests to a remote database server (DB2B) using DRDA. This configuration is shown in Figure 3-7 on page 34.

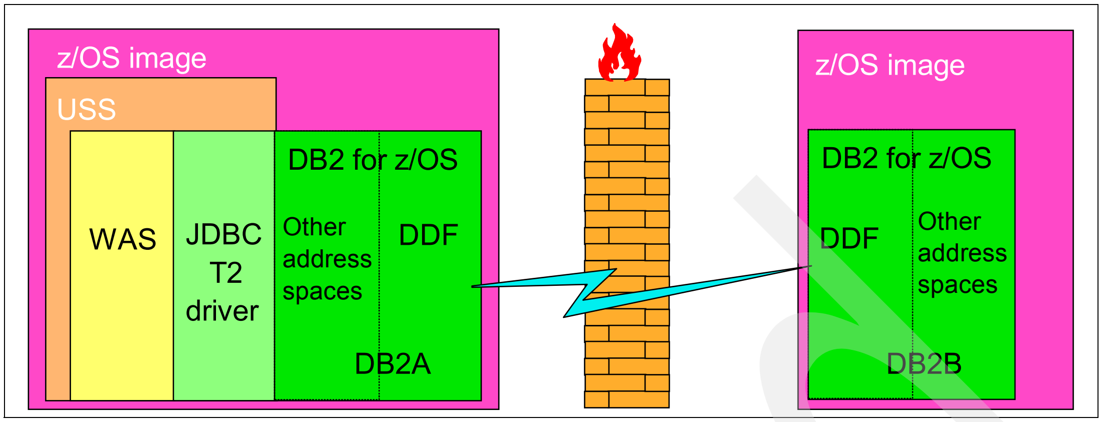

Figure 3-7 DB2 for z/OS and OS/390 as DRDA AR

In this configuration, DB2A does not have any databases that are accessed by applications running on the WebSphere Application Server. All SQL requests are routed through the local DB2A, but are accessing data on the remote DB2B through DRDA.

In this configuration, we need the DRDA Application Requester functionality of DB2A, to access the data residing on DB2B. This is because the current (non-Universal) JDBC driver for z/OS and OS/390 is a Type 2 driver and cannot directly access a remote DB2 system (DB2B). (This also applies when using the Universal Driver with type 2 connectivity.)

From the Java application's point of view, the local DB2 is "transparent".The application will connect directly to the remote DB2 system. However, the local DB2 has to exist, because its services as a DRDA Application Requester (AR) are needed to be able to connect to the remote DB2 system.

Note that from a performance standpoint this is not the ideal configuration, as there is the additional overhead of having to route every SQL request over DRDA over (most likely) TCP/IP to a remote DB2 system.

3.2.6 IBM z/OS Application Connectivity to DB2 for z/OS and OS/390

z/OS Application Connectivity to DB2 for z/OS and OS/390 is a no-charge, optional feature of DB2 Universal Database Server for z/OS and OS/390, Version 7, and DB2 for z/OS Version 8.

This feature consists of a component known as the DB2 Universal Database Driver for z/OS, Java Edition. This is a pure Java, Type 4 JDBC driver designed to deliver high performance and scalable remote connectivity for Java-based enterprise applications on z/OS to a remote DB2 for z/OS database server. The driver:

- Supports JDBC 2.0 and 3.0 specification and JDK V1.4 to deliver the maximum flexibility and performance required for enterprise applications
- Delivers robust connectivity to the latest DB2 for z/OS and WebSphere Application Server for z/OS
- Provides support for distributed transaction support (two-phase commit support)
- Enables custom Java applications that do not require an application server to run in a remote partition and connect to DB2 z/OS

With IBM z/OS Application Connectivity to DB2 for z/OS and OS/390, shown in Figure 3-8, you no longer need the local DB2 (DB2A) on the same machine as your a WebSphere Application Server, as in Figure 3-7 on page 34. This no-charge feature of DB2 for z/OS and

OS/390 V7 and DB2 for z/OS V8 provides a Type 4 JDBC driver that supports two-phase commit. (This driver is sometimes [unofficially] called the Type 4 XA driver.) Your Java applications running inside the a WebSphere Application Server talk to the (Universal) Type 4 JDBC driver that supports two-phase commit, and the driver talks directly to the remote database server (DB2B) through DRDA. The Universal Type 4 driver implements DRDA Application Requester functionality. You do not need to buy a DB2 license for DB2A when using the DB2 Universal Database Driver for z/OS, Java Edition. It is sufficient to have a DB2 license for DB2B.

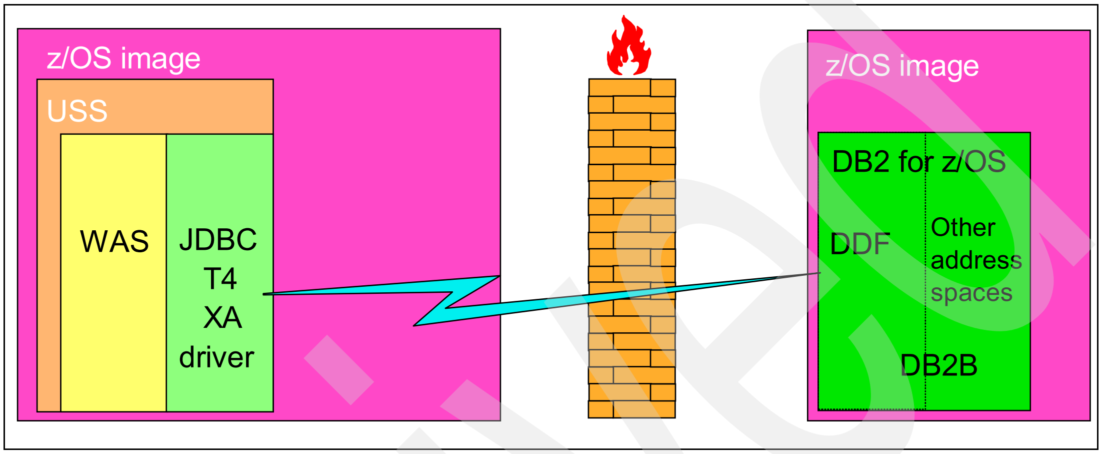

Figure 3-8 z/OS Application Connectivity to DB2 for z/OS and OS/390

You are only authorized to use this feature when connecting an application running on z/OS or OS/390, to DB2 UDB for z/OS and OS/390 V7 or DB2 for z/OS V8, running in a separate LPAR on the same server as the application, or running on a different z/OS or OS/390 server (but only between z/OS and/or OS/390 environments).

The z/OS Application Connectivity to DB2 for z/OS and OS/390 feature provides connectivity from a z/OS or OS/390 remote partition or system only. For access from any other operating system or platform, including z/Linux, to DB2 for z/OS and OS/390, you must obtain a separate license of the edition of DB2 Connect that is appropriate for your environment.

This feature can only be ordered for DB2 UDB for z/OS and OS/390 V7, or DB2 for z/OS V8. It ships as a separate FMID, HDDA210.

DB2 UDB for OS/390 and z/OS V7 servers do not have built-in support for distributed transactions that implement the XA specification. When accessing a DB2 V7, the DB2 Universal JDBC Driver supports distributed transactions (two-phase commit), but has to emulate that support. You have to run an additional setup step to create the SYSIBM.INDOUBT table and its package (via the DB2T4XAIndoubtUtil utility program).

DB2 UDB for z/OS Version 8 has native XA two-phase commit support in DRDA.

For additional information, see the applicable announcement letter. For the United States, see IBM Software announcement letter 203-351, dated December 16, 2003.

For more information, see *DB2 for z/OS Application Programming Guide and Reference FOR JAVA™ Version 8,* SC18-7414-**01**, and the Redbook *DB2 for z/OS and WebSphere: The Perfect Couple*, SG24-6319.

3.3 Developing a Java application using JDBC

Let us now look at how to develop a Java application that wants to talk to a database through JDBC. In this section, we focus on the connection part of the process: How to connect to the DB2 system from a Java application. More details on how to actually retrieve and update data, is provided throughout the rest of this publication. This section also applies to SQLJ programs, as SQLJ uses JDBC to establish a connection to a database.

3.3.1 Connecting to a database

Connecting to the database means you make the connection between the application and the driver that you will use in the program.

There are two methods that are used to set the driver and database for the connection:

- The *DriverManager* interface
- The *DataSource* interface

When using the *DriverManager* interface the names of both the driver and database are coded directly into the method calls within the application code.

Using a *DataSource* object instead of DriverManager removes this direct coding of driver and database names from the application. This information is stored in an external object that is set up and administered independently from the application that the application accesses through the use of a logical name. The differences between both interfaces are where the data source definitions can be defined, and the levels of portability that it provides.

A thing to remember is that the term “database” means different things on different platforms.

- With DB2 for z/OS and OS/390, when you are connecting to a database, you are actually connecting to a whole DB2 subsystem (or a data sharing group), which will contain multiple databases. Depending on the security within that subsystem, when you connect you can potentially access all databases, tables and views in that subsystem.
- With a DB2 for Windows or Unix database, you are connecting to a specific database, and you will be able to only access the tables and views within that database.

This is illustrated in Figure 3-9.

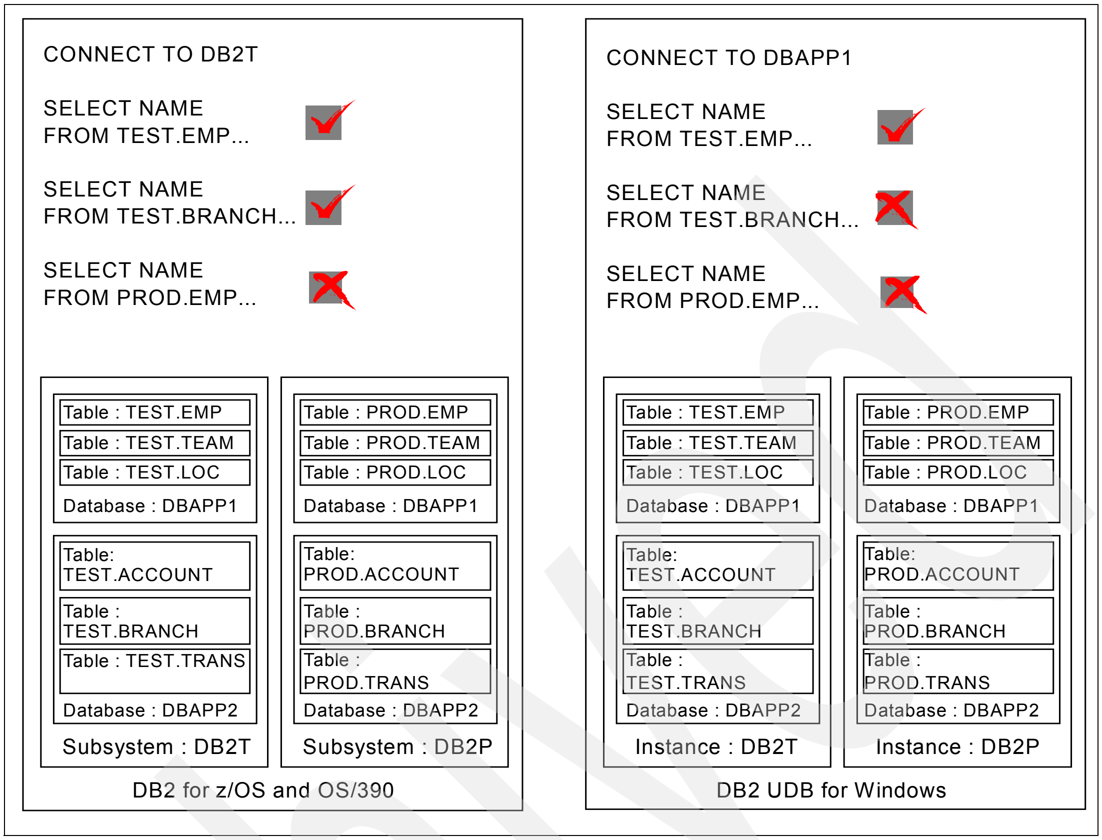

Figure 3-9 Scope of connection

3.3.2 Using the DriverManager interface

This is the older method of connecting to a data source, and is available in all flavors of JDBC. When using the DriverManager interface, the application must define specific JDBC driver classes and database names. This limits portability. If an application is in development and requires a local database for testing purposes, when the application is migrated to the production environment, the code has to be changed to reflect the change of drivers and the database connection.

Loading the DB2 driver

To use the DriverManager interface the program first loads the required JDBC driver by invoking the `Class.forName` method. (The Class class is in the java.lang package, which Java loads automatically, and therefore no import statement is required when using this method.) The method can be coded as shown in Example 3-1 on page 37.

Example 3-1 Loading the DB2 JDBC driver

```
try {
  // Load the DB2 driver
  Class.forName("com.ibm.db2.jcc.DB2Driver");
} catch (Throwable e) {
  e.printStackTrace(e);
}
```

What happens here is that the Java Virtual Machine brings in the code for that particular JDBC driver. The driver automatically registers itself with the JDBC runtime and can now be used to connect to a database.

This code sample loads the (JCC) Type 4 JDBC driver, which implements JDBC 3.0 features. The actual driver that will be used depends on how your CLASSPATH is set up at the time when the application is run. See Table 3-1 on page 26 for the combinations of classes referenced in the code, and which file needs to be included in the CLASSPATH to designate which class is actually used.

The DB2 Universal Driver for SQLJ and JDBC Type 4 driver is the driver that we will be using throughout most of this publication, although we do use the Type 2 drivers where specific functionality is not available in the Type 4 driver. Remember that when using the Type 4 driver db2jcc.jar must be added to the Java CLASSPATH. If other JDBC classes are present in the classpath, then this jar file must come first; otherwise you get an error if your program contains references to new functions that are only available in this new driver. For example, if you use the Type 4 URL to connect, and the old (non-JCC) Type 2 driver class in your CLASSPATH, the program will fail, as Type 4 connections are not available from that driver.

Establishing a connection

Once the driver is loaded a connection to the data source needs to be invoked. This is done by using the `DriverManager.getConnection` method, which comes in three forms:

- `getConnection(String url);`
- `getConnection(String url, user, password);`
- `getConnection(String url, java.util.Properties info);`

JDBC defines the format of the URL as:

`jdbc:`*`driverselection`*`:`*`database`*

Where *driverselection* represents the database-specific type designator (such as db2). What the JDBC runtime does is to examine all drivers that have been loaded (and therefore registered with the JDBC runtime), asking each of them in turn whether they accept the *driverselection* part. The first driver to answer *true* is used for the connection, and is passed the *database* part, which again has a driver-specific syntax.

This way, several JDBC drivers can simultaneously loaded into a Java Virtual Machine, and the JDBC runtime can determine which one to use based on the connection URL. Table 3-2 shows the different formats to use when connecting to a DB2 for z/OS and OS/390 data source.

Table 3-2 JDBC Drivers and URL formats

Driver class	URL format	Driver type
Com.ibm.db2.sqlj.jdbc.DB2SQLJDriver	`jdbc:db2os390sqlj:`*`database`*	2
COM.ibm.db2.jdbc.app.DB2Driver	`jdbc:db2:`*`database`*	2
com.ibm.db2.jcc.DB2Driver	`jdbc:db2:`*`database`*	2
com.ibm.db2.jcc.DB2Driver	`jdbc:db2://`*`server`*`:`*`port`*`/`*`database`*	4

Driver class	URL format	Driver type
Notes: ► *database* represents the name of the target subsystem or DB2 location name. ► *server* represents the TCP/IP address or host name of the target system. ► *port* represents the SQL port number used by the DB2 server. It can be omitted if this server uses the DB2 default port (446).		

To connect to a database using the Type 4 driver, the code is shown in Example 3-2.

Example 3-2 Connecting using the Type 4 driver

```
Connection con =  DriverManager.getConnection("jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y",
                                             "sflynn",
                                              "girliel");
```

The DriverManager interface is part of the java.sql package.

As we have mentioned earlier, the DriverManager interface lacks portability, as the application needs to specify specific JDBC drivers, URLs and databases in the code. Using the DB2 Universal Driver for Java Common Connectivity has simplified this, as you can point to the same DB2 OS/390 or z/OS database from both the Windows platform as well as z/OS or OS/390. However, with other drivers, code must be changed directly before it can be executed on the mainframe.

In most DB2 installations, different databases (and subsystems) are used to store the data in a development versus a production environment. For example, the test data is stored in a test subsystem called DB2T, while the production database is run in the production subsystem DB2P. In this case, when an application is migrated from test to production, the URL will have to be changed for all drivers when the DriverManager interface is used.

A way to avoid this problem is to externalize all references to the driver and connection, and load it into the application at execution time. You can do this by creating platform specific property files, which contain entries for the driver and database, and read these values in at runtime. Java provides a class to read the properties file and access these entries (the `java.util.Properties` class). It can be used by all types of Java applications. We show an example of this technique in "Creating a test driver" on page 204.

For stand-alone Java programs, another way to externalize these values is to specify them as input arguments to the program in the command line.

For server Java programs this option is not available. The driver and database details can be set in the initialization parameters for the application, which can be accessed using the getInitParameters method within the code.

The best way to maintain database connectivity is to avoid using the DriverManager altogether. This is achieved by using the DataSource interface, which allows system-administered and secure definitions of the data connections between an application and its data sources. If you have the technology to support this method (JDBC 2.0 compliant drivers), then DataSource connections should be the method of choice.

3.3.3 Connecting using the DataSource API

To avoid the problems that the DriverManager has with portability issues, another method of storing data source information is defined in JDBC 2.0. This method uses a new class called the *DataSource*. With this object, all data pertaining to the connection is stored in a system managed area that is accessed through a logical name. The thing to remember is that this *DataSource* object is persistent, and once it has been created, it is maintained as a system object.

Once created, the DataSource object is brought into the application referring to its logical name. The beauty of this method is that you can have DataSource objects with the same logical names defined on a different platform that contains platform-specific information, and the program will automatically plug into it when it is moved to that platform.

Setting up a DataSource

The DataSource object is defined and managed within the application server in a normal development platform where development work is done on a client platform, such as Windows, with the code ultimately being deployed on the z/OS or OS/390 platform.

An important issue to note is that as the support for data source objects normally comes from the application server (such as WebSphere). For an example on how to set up a data source, see “Setting up a data source in WebSphere Studio” on page 258.

The program that creates and manages a DataSource object also uses the Java Naming and Directory Interface (JNDI) to assign a logical name to the DataSource object. The JDBC application that uses the DataSource object can then refer to the object by its logical name, and does not need any information about the underlying datasource (see next section for details). In addition, your system administrator can modify the data source attributes, and you do not need to change your application program.

To learn more about using WebSphere to deploy DataSource objects, go to:

http://www.ibm.com/software/webservers/appserv/

Connecting with a DataSource object

As mentioned before, the advantage of using a DataSource is that you avoid having to hard code drivers and URLs in the program, and that those values are stored in the data source on the server.

When connecting using a DataSource, the first thing you must code in your program are the import statements for the class that support data source objects. These are:

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;
```

The first two import statements define the classes that support retrieving the data source values from the server, while the last one is the JDBC class for data source object support.

The code for a connection using data sources for retrieving the values stored in the server uses what is called a *context*. This returns the actual values of the data source.

So, the first thing to do is to define the context you are going to use:

```
Context ctx;
```

Then declare the connection itself:

```
Connection conn;
```

Now you must open the data source object and return the values that are needed to make the connection to the database:

```
ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/DB7Y");
```

Now the values relating to the data source have been acquired by the program and the connection can be made. To do this you code the following statement:

```
conn = ds.getConnection();
```

Or:

```
conn = ds.getConnection(userid, password);
```

And the connection is made. The first version of the `getConnection()` method uses the default user ID and password that is stored in the server. On the Windows version of WAS, these values are stored in the data source object itself. With WAS for z/OS and OS/390 no option is provided to store these values in the data source, so the default user ID and password are stored in the server itself.

3.4 Accessing data using SQLJ

SQLJ is a series of specifications for ways to use the Java programming language with SQL. It was developed by IBM, Oracle and Tandem to provide an alternative to the dynamic JDBC specification. SQLJ is not part of J2EE, but is part of the SQL-1999 ISO/ANSI standard.

SQLJ consists of two parts:

- ISO/IEC 9075 Part 10: Object Language Bindings (SQL/OLB). This is the specification for embedded SQL in Java, and is what we discuss in the later chapters of this publication.
- ISO/IEC 9075 Part 13: Routines and Types Using the Java Programming Language (SQL/JRT). This part is the specification for SQL routines using Java.

In this publication, we focus on the support of embedded SQL in Java applications and Servlets.

There are three different forms of the SQLJ statement:

- Executable statements
- Iterator declarations
- Connection declarations

Refer to Chapter 10, “SQLJ tutorial and reference” on page 175, which discusses the different types of SQLJ statements and provides code examples.

For programmers in other mainframe languages, such as COBOL and PL/I, embedded static SQL is almost the default way of coding DB2 database access. For programmers on other platforms, it is far more likely that they will be coding using dynamic statements utilizing ODBC/JDBC methods.

The major difference between dynamic JDBC Java programs and (static) SQLJ programs, from a development point of view, is that SQLJ programs have to be preprocessed prior to execution. This is due the way SQL statements are coded in SQLJ programs. Unlike JDBC, which is a call-level interface (an API), SQLJ is a language extension. When an SQLJ program has been coded, it is first run through the SQLJ translator, fittingly called sqlj. This takes the SQLJ statements and replaces them with valid Java methods. The SQL statements are placed in a file name *ClassName*_SJProfile0.ser while the Java code is placed in a standard Java file called *ClassName*.java. The program file can now be compiled and run (the translator compiles the program automatically unless you use a command line switch to suppress it).

However, besides the mandatory translation step, you can (and should) execute an optional customization step. Not all DBMS systems supporting SQLJ also support customization, but DB2 does. During customization against a DB2 system, the SQL statements in the .ser files are bound into packages, and they will execute as true static SQL statements at runtime. In case the DBMS does not support customization, or customization has not been done, the translated SQLJ statements are executed through JDBC as dynamic SQL.

Important: The real strengths of SQLJ come only into play through the customization process. Do not expect better performance of SQLJ programs over equivalent JDBC programs if no customization has taken place. In fact, uncustomized SQLJ will rather run slightly slower because of some additional overhead in the SQLJ runtime.

For a more extensive discussion of the methods for SQLJ program preparation, and why the customization step is so important, please refer to Chapter 9, “The SQLJ program preparation process” on page 149.

3.5 Using JDBC or SQLJ

Following are some good reasons to consider using SQLJ, as well as some reasons why JDBC may be more appropriate.

3.5.1 SQLJ is easier to code

The first advantage of SQLJ over JDBC is that SQLJ is easier to code, to read, and to maintain. This is an effect of SQLJ being not an API, but a language extension, providing for better integration of the SQL code with the Java code. The developer can concentrate on the logic of individual SQL statements without having to worry about wrapping them in API calls. This simplicity is helped by the ease by which host variables are defined, maintained and accessed within an SQLJ program.

As SQL is coded in purely SQL syntax, without the need to wrap them in a Java method, the programs themselves are easier to read, making them easier to maintain. Also, since some of the boilerplate code that has to be coded explicitly in JDBC is generated automatically in SQLJ, programs written in SQLJ tend to be shorter than equivalent JDBC programs.

We illustrate the easier SQLJ syntax in the following examples, showing a multi-row query, a single-row query, and an INSERT statement.

Example 3-3 on page 43 shows a multi-row query. The amount of coding is similar with JDBC and SQLJ. Note, however, that the binding between statement and host variables in SQLJ is much tighter than between parameter markers and the `setBigDecimal()` methods in JDBC. Also, JDBC uses statement handles that must be explicitly closed when you are done with the statement. In SQLJ, the translator automatically generates the cleanup code for you. (Iterators must still be closed explicitly, of course.)

Example 3-3 JDBC vs. SQLJ: Multi-row query

```
PreparedStatement stmt =
  conn.prepareStatement(
    "SELECT LASTNAME"
  + "     , FIRSTNME"
  + "     , SALARY"
  + "  FROM DSN8710.EMP"
  + " WHERE SALARY BETWEEN ? AND ?");

stmt.setBigDecimal(1, min);
stmt.setBigDecimal(2, max);

ResultSet rs = stmt.executeQuery();
while (rs.next()) {
   lastname = rs.getString(1);
   firstname = rs.getString(2);
   salary = rs.getBigDecimal(3);
  // Print row...
}
rs.close();
stmt.close();
```

```
EmployeeIterator iter;
#sql [ctx] iter = {
  SELECT LASTNAME
       , FIRSTNME
       , SALARY
    FROM DSN8710.EMP
   WHERE SALARY BETWEEN :min AND :max
};

while (true) {
  #sql {
    FETCH :iter
     INTO :lastname, :firstname, :salary
  };
  if (iter.endFetch()) break;
  // Print row...
}
iter.close();
```

In Example 3-4, we compare how to code a single-row query, that is, a query returning exactly one row of data. In JDBC, we have to open a result set, advance it to the next (and only) row, and retrieve the values using `getXxx()` methods. Also, we have to check if exactly one row has been found. In SQLJ, on the other hand, we can use the SELECT INTO syntax; an `SQLException` will be thrown if no row or more than one row was found.

By the way, the SQLJ version is more efficient as well. JDBC has to make four calls to DB2 (prepare statement, fetch row, fetch row, close statement), whereas the SQLJ version only has to do one single SELECT INTO call.

> **Note:** The SQLJ version will only be more efficient when the program has been customized and bound. If it is running uncustomized, it will emulate SELECT INTO by using result sets under the cover, just like the JDBC version.

Example 3-4 JDBC vs. SQLJ: Single-row query

```
PreparedStatement stmt =
  conn.prepareStatement(
    "SELECT MAX(SALARY), AVG(SALARY)"
  + "  FROM DSN8710.EMP");
rs = statement.executeQuery();
if (!rs.next()) {
  // Error -- no rows found
}
maxSalary = rs.getBigDecimal(1);
avgSalary = rs.getBigDecimal(2);
if (rs.next()) {
  // Error -- more than one row found
}
rs.close();
stmt.close();
```

```
#sql [ctx] {
  SELECT MAX(SALARY), AVG(SALARY)
    INTO :maxSalary, :avgSalary
    FROM DSN8710.EMP
};
```

As a last example, consider the INSERT statement in Example 3-5. Again, the SQLJ code is easier to read and to maintain since the boilerplate code for supplying parameters to the statement, and for resource cleanup, is generated for you by the SQLJ translator.

Example 3-5 JDBC vs. SQLJ: INSERT statement

```
stmt = conn.prepareStatement(
   "INSERT INTO DSN8710.EMP ("
  + "  EMPNO, FIRSTNME, MIDINIT"
  + ", LASTNAME, HIREDATE, SALARY) "
  + "VALUES (?, ?, ?, ?, CURRENT DATE, ?)"
);
stmt.setString(1, empno);
stmt.setString(2, firstname);
stmt.setString(3, midinit);
stmt.setString(4, lastname);
stmt.setBigDecimal(5, salary);
stmt.close();
```

```
#sql [ctx] {
  INSERT INTO DSN8710.EMP (
      EMPNO, FIRSTNME, MIDINIT
    , LASTNAME, HIREDATE, SALARY)
  VALUES
    ( :empno, :firstname, :midinit
    , :lastname, CURRENT DATE, :salary)
};
```

Also note that people with a (static) embedded SQL programming background in for example COBOL, will find it very easy to start using SQLJ, as iterators and SELECT INTO constructs look very much like those in embedded SQL.

3.5.2 SQLJ catches errors sooner

Not only is SQLJ typically more concise and easier to read than JDBC, it also helps you to detect errors in your SQL statements earlier in the program development process.

JDBC is a pure call-level API. This means that the Java compiler does not know anything about SQL statements at all—they only appear as arguments to method calls. If one of your statements is in error, you will not catch that error until runtime when the database complains about it.

SQLJ, on the other hand, is not an API but a language extension. This means that the SQLJ tooling is aware of SQL statements in your program, and checks them for correct syntax and authorization during the program development process.

It also enforces *strong typing* between iterator columns and host variables. In other words, it prevents you, for example, from assigning a numeric column to a String host variable.

Common errors that will be caught earlier with SQLJ, but will only be detected at runtime by JDBC, include:

- Misspelled SQL statements (for example, INERT instead of INSERT)

  The SQLJ translator will catch and report this error. However, the translator does not parse the entire SQL statement, so most syntax errors will only be detected by the profile customizer.

- No parameter supplied for parameter marker

  Consider Example 3-3 on page 43. If you forgot to supply a value for one of the two parameter markers in JDBC, a runtime error will occur. In SQLJ, there are no parameter markers, rather, the host variables are embedded in the statement.

- Misspelled table name

  A misspelled table name will not be caught by the SQLJ translator (after all, it cannot know if a table does exist or not); however, the profile customizer will complain unless it was invoked with online checking disabled. (For a discussion of profile customizing and online checking, see Chapter 9, “The SQLJ program preparation process” on page 149.)

- Not all columns retrieved

  Assume that you add another column to the SELECT list in Example 3-3 on page 43 but forget to retrieve the corresponding column in the loop that processes the result set. This cannot happen with SQLJ, since the number of host variables in the FETCH statement (and the number of columns in the SELECT list) must match the number of columns in the iterator declaration.

3.5.3 SQLJ is faster

In most circumstances, an SQLJ application runs faster than its JDBC equivalent, provided it has been customized against the database to use static SQL.

To execute an SQL statement, the following steps have to be performed:

- The SQL statement is parsed into an internal form.
- DB2 checks that the user ID has sufficient authority to execute the statement.
- DB2 calculates an access path.

With dynamic SQL, these steps are performed at runtime, and they have to be executed each time the program runs (provided you are not using the dynamic statement cache). With customized SQLJ, on the other hand, all three steps are done at development time. DB2 only has to check whether the user running the program is authorized to execute the package that had been created for the program. This is illustrated in Figure 3-10.

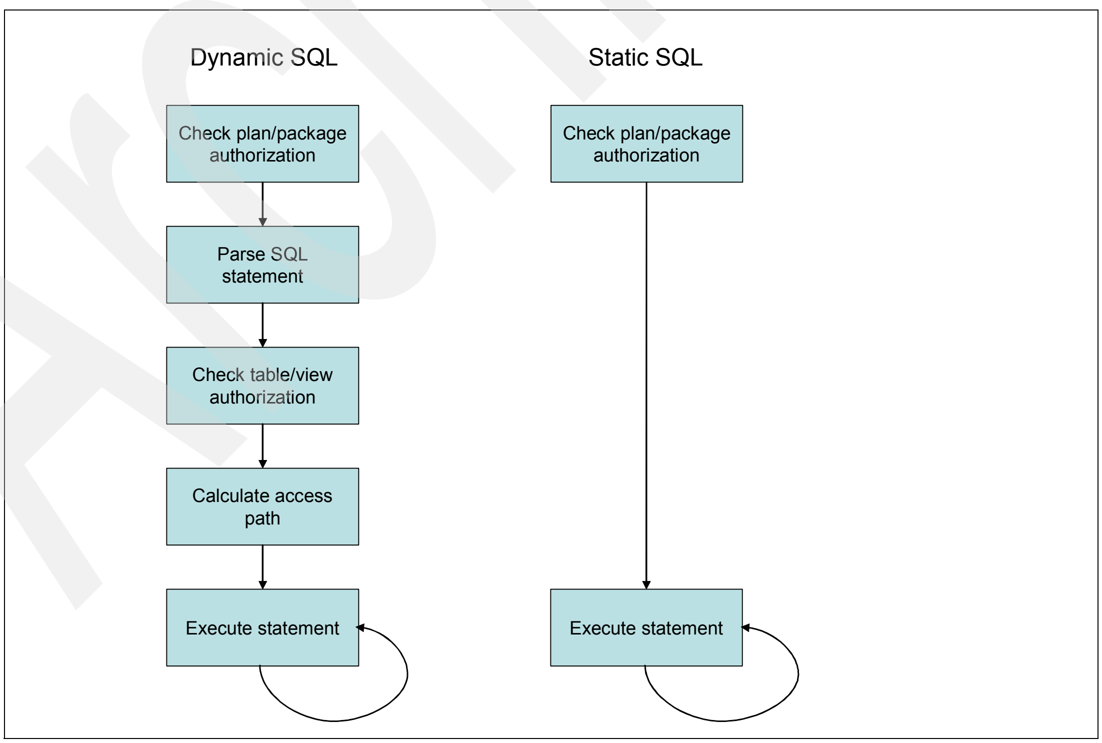

Figure 3-10 Execution of dynamic vs. static SQL statements

On the other hand, as already pointed out, the customize and bind steps are optional. When you run uncustomized, the SQLJ runtime uses dynamic SQL under the cover. This allows you to start prototyping your SQL statements very early on during the development process, running uncustomized. Once you are moving on to more detailed testing, you can customize and run as static SQL. Thus, SQLJ combines the best of both worlds: Easy development and good performance.

3.5.4 SQLJ provides better authorization control

Perhaps even more important than the performance aspect of the static SQL model is the security aspect. If you use dynamic SQL, the user ID under which you run your program must have all privileges required by the code. For example, if the program wants to update rows in the DSN8710.EMP table, and is running under your user ID, your user ID must have UPDATE authority on that table (or belong to a group that has). If any other user wants to run your program, he or she must also be given that authority.

There are two problems associated with this:

- Managing these authorizations when a large number of people need to access the data, as they all need to be granted access, is a cumbersome and error-prone task. (This can be partially solved by grouping users together in groups and granting access to the group instead of each individual user.)
- The second problem with this is that once a user has the authority to update a table, he can do so not only using the application he wanted to use in the first place, but also, for example, with interactive tools. Obviously, this can be a big exposure. The application may be coded to observe the company's business rules, but the interactive user can make any change he or she wants, bypassing the company's business rules enforced through the application.

The use of static SQL solves this problem. With static SQL, a user can be authorized to run a program containing statements (such as UPDATEs) for which that user himself does not have sufficient privileges. In other words, the point of control is not persons, but programs.

This is illustrated in Figure 3-11 on page 47, Figure 3-12 on page 47, and Figure 3-13 on page 48.

In Figure 3-11 on page 47, user DIANE is a member of the PAYROLL group, which has SELECT but no UPDATE privilege on the EMP table. Diane wants to run a JDBC application that needs to update the EMP table. Her attempt to update the EMP table through the JDBC application will fail since the application runs using Diane's insufficient privileges. Diane does, however, have SELECT privilege on the table since she is a member of the PAYROLL group. Therefore, she can SELECT from the table using interactive tools such as QMF™.

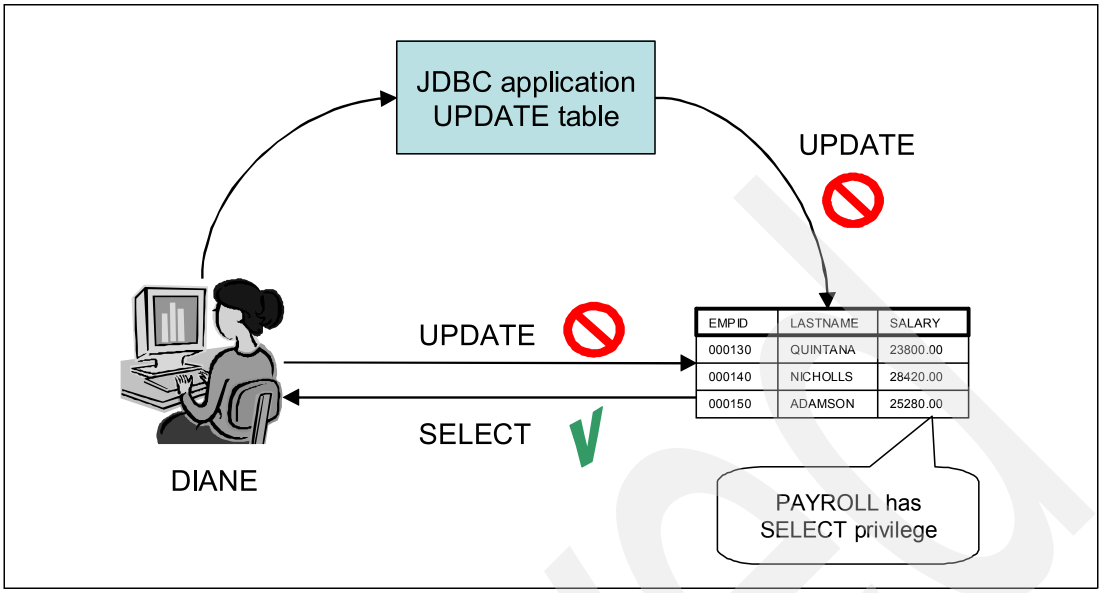

Figure 3-11 Dynamic SQL, with SELECT but no UPDATE privilege

In order for Diane to be able to successfully run the JDBC application, she must be given UPDATE privilege on the table, either by granting her the privilege individually, or by granting it to the PAYROLL group (Figure 3-12). The problem with this is that Diane can now update the EMP table directly, for example, by using QMF, with obvious security implications.

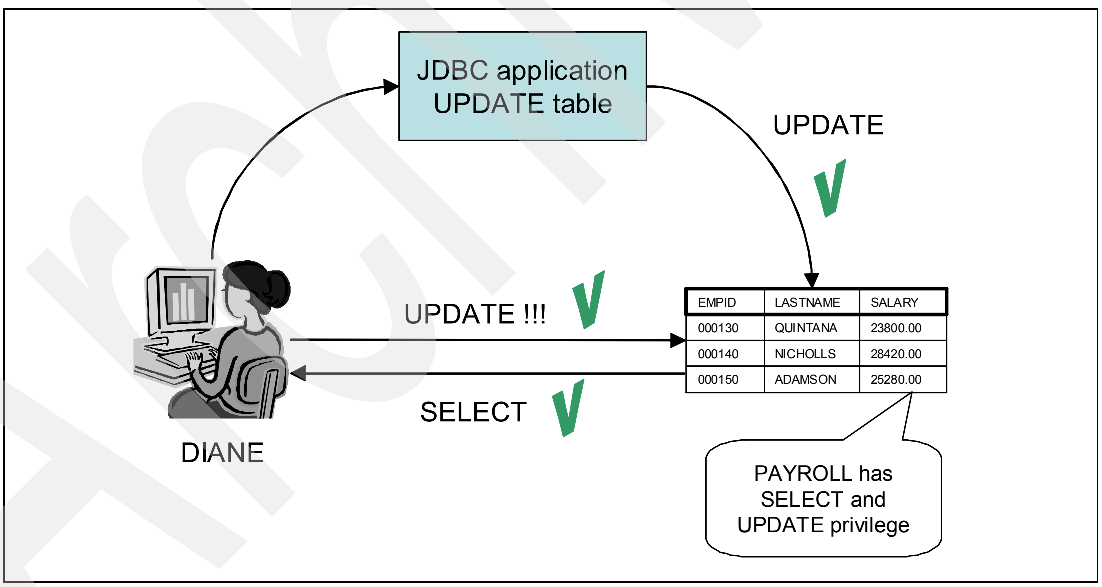

Figure 3-12 Dynamic SQL, with SELECT and UPDATE privilege

What we really want is to enable Diane to successfully run the application *without* giving her direct UPDATE privilege. This can be done with static SQL, as shown in Figure 3-13 on page 48.

User FRED, the database administrator, who has UPDATE privilege on the table, binds a package for the SQLJ application, and grants EXECUTE privilege on the package to the PAYROLL group. Since Diane is a member of the PAYROLL group, she can run the application (and execute the package), which can then update the database using Fred's privilege. Diane cannot, however, update the table directly through, for example, QMF.

In this configuration, the trusted entity is no longer a user, but an application.

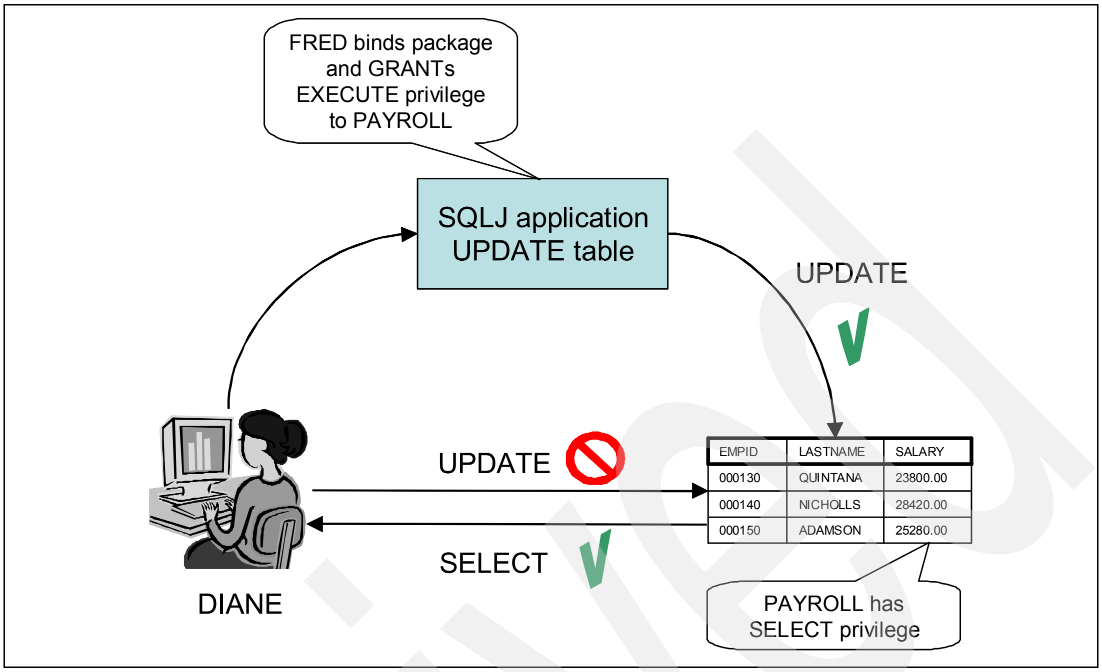

Figure 3-13 Static SQL, no UPDATE but EXECUTE privilege

Note: When using the Type 2 driver, life is slightly more complicated. In that case, Diane needs EXECUTE authority on the plan to execute it (that includes the package in its PKLIST) and not EXECUTE authority on the package. But as before, the trusted entity is no longer a user, but an application.

3.5.5 SQLJ is more predictable and reliable

The key here is that you have static SQL statements that are bound into packages. The access paths for the statements are locked in during package bind. They will not change on your production system unless you recompile the program or rebind.

This means you can use normal change control procedures to manage the Java programs, and can reasonably expect that the programs will run and perform the exact same way until the next time you make a concious change (rebind or recompile).

3.5.6 SQLJ allows for better monitoring

With SQLJ, you get much better system monitoring and performance reporting. Static SQL packages give you the names of the programs that are running at any given point in time. This is extremely useful for studying CPU consumption by the various applications, locking issues (such as deadlock or timeout), etc.

3.5.7 SQLJ Tooling

WebSphere Studio Application Developer (WSAD) V5.1 has full support for SQLJ, including native editor support, debugger support, integrated EJB CMP support and integrated SQLJ translation. WSAD also supports profile customization and binding to generate static SQL for maximum performance. The usage of the SQLJ editor is shown in Chapter 8, “Getting started with SQLJ" on page 141. We also demonstrat the use of the new SQLJ customization support in WSAD in "Preparing SQLJ programs to use static SQL through WSAD" on page 157.

3.5.8 Use JDBC for flexible SQL statements

If there is a requirement for ad-hoc SQL statements, then JDBC is usually a better, if not the only, option. For example, if your program is in response to a Web page that allows a user the option of selecting many different search parameters, the resulting SQL will have to be versatile enough to handle any of these combinations. In this situation it is better to build the SQL query at runtime and run it using JDBC.

The SQLJ alternative would be to either build an SQL statement for each possible combination, or write a query that has slots for all possible fields and then use `like` match string values and number ranges for numeric fields. Both these solutions are rather dire and should be avoided.

A better programmatic solution is to use SQLJ for all stable SQL while, from within the same program, coding all changing SQL in JDBC methods.

3.5.9 SQLJ/JDBC interoperability

JDBC and SQLJ are complementary technologies, and can be mixed and matched within an application, for example:

- SQLJ and JDBC can share the same connections.
- SQLJ iterators and JDBC result sets can be converted to each other.

This is explained in more detail in "Interoperability between JDBC and SQLJ" on page 198.

In fact, there is no SQLJ equivalent to the PREPARE, EXECUTE and DESCRIBE statements used in other programming languages with embedded SQL support, since this functionality is already covered with JDBC.

3.6 Summary

Table 3-3 summarizes the differences between JDBC and SQLJ that we discussed in the previous sections.

Table 3-3 Differences between JDBC and SQLJ summarized

JDBC	SQLJ
Dynamic SQL	Static SQL
Call level interface (API)	Embedded SQL (language extension)
No precompile or binding necessary	Precompile necessary; bind optional but strongly recommended
Syntax checks at runtime	Syntax checks at compile time
Authorization checks at runtime	Authorization checks at bind time
Access path calculation at runtime	Access path calculation at bind time
User-based authorization model	Program-based authorization model
Can construct SQL at runtime	All SQL hardwired

JDBC	SQLJ
Standard part of the Java platform	Not a standard part of Java, but an ISO standard (ISO/IEC 9075:10-2000)

Part 2

Prerequisites and setup

In this part we discuss how to set up your Java and DB2 environment properly to be able to fully enjoy this brave new world.

As we do most of our development through WebSphere Studio Application Developer (WSAD), we also discuss the WSAD and Windows setup.

4

Products and levels - Now and soon

This chapter provides a brief overview of the products we used during the development of this publication, as well as which versions of the software products we used.

4.1 Products and levels

With the widespread demand for Java functionality, advances arrive on a frequent basis. This means that the Java aspect of almost every product is in constant flux. Few enhancements arrive on product release boundaries. This is a fact of life and will remain so.

We list in this section the products we used (“Now” on page 54) when we wrote this publication. We are aware of some future enhancements to the products we used and list them for you in “Soon” on page 54. It is likely that some of them may arrive sooner or later than we think, but we want you to understand the implications.

4.1.1 Now

At the time of writing this publication:

- DB2 UDB V8 for Linux, Unix and Windows, FixPak 2 was available at the time we wrote this publication. It introduces the Type 2 Java Universal Driver. The Type 4 SQLJ and JDBC Driver of the Universal Driver family was introduced at DB2 for LUW V8 GA time. In the meantime FixPak 3 became available in August 2003 and testing for FixPak 4 is underway.
- DB2 for OS/390 and z/OS V7 at RSU0306. Service for the following APARs necessary to use the DB2 Universal Driver for SQLJ and JDBC was applied:
  - PQ62695
  - PQ72453

  For details, see “Required service” on page 58.
- z/OS 1.4 at RSU0306.
  - Java SDK 1.3.1
- WebSphere Application Server V5.0.0 for z/OS Build level W501000 9/15/2002.
- WebSphere Studio Application Developer V5.1 (WSAD) on Windows. Not yet generally available when we wrote this publication. We downloaded it from an IBM internal site at this level:
  - 5.1.0 - Based on 20030720_2137 Build
  - With the initial SQLJ tooling support added in WSAD V5.1, it is no longer necessary to leave WSAD to run the SQLJ translator or to perform SQLJ profile customization in the DB2 Command Window for the BMP beans, session beans and Servlets.
- Java Runtime Environment (JRE) 1.3, supplied with DB2 UDB V8 and WSAD V5.1.
- Java 2 Runtime Environment, Standard Edition (build 1.3.1) Classic VM (build 1.3.1, J2RE 1.3.1 IBM OS/390 Persistent Reusable VM build cm131s-20030913 (JIT enabled: jitc)).

4.1.2 Soon

Soon:

- DB2 for OS/390 and z/OS V7:
  - Universal Driver (Type 4) planned for first quarter of 2004
  - Universal Driver (Type 2) planned for first quarter of 2004

> **Note:** The DB2 Universal Driver for SQLJ and JDBC will be made available for DB2 for z/OS and OS/390 Version 7 through the PTF for APAR PQ80841. This APAR was still open at the time of writing of this publication.

When both occur, the SQLJ and JDBC drivers will be common with DB2 UDB V8. The implication for developers who wish to deploy SQLJ programs to the DB2 for OS/390 and z/OS V7 platform is that there is a common SQLJ profile customizer (**db2sqljcustomize**), and the serialized profiles are portable across the DB2 family. During the development of this publication we used early versions of both the T4 and T2 driver for z/OS. Actually, since the T4 driver is a pure Java driver, you can copy it, together with the required license files, from your DB2 for LUW installation into a USS directory on your z/OS system (which we actually did, and it works fine, at least for all our tests).

– Developers can deploy on any server platform without running the profile customizer on the target system.

– The .ser file contains information needed for all BIND operations, without having to recustomize on each BIND. Note that DBRMs will no longer be produced.

- DB2 UDB for z/OS V8 will have the SQLJ and JDBC Universal Driver incorporated when it becomes generally available.
- The current JCC T4 driver does not support 2-phase commit. This will change in the near future. Then, the T4 driver will also be able to do 2-phase commit. This functionality will be implemented in both the V7 and V8 of DB2 for z/OS.

5

Setup

This chapter describes the setup necessary to execute the samples for the following products:

- DB2 for OS/390 and z/OS V7
- Workload Manager (WLM)
- Unix System Services
- DB2 Universal Driver - Setup for a Windows environment
- WSAD setup
- WAS for z/OS V5 data source setup

5.1 DB2 for OS/390 and z/OS V7

Our samples use the employee sample table (DSN8710.EMP). Make sure you have been granted INSERT, SELECT, UDPATE, and DELETE access, or grant access to PUBLIC. The DB2 sample database is created by running the DSNTEJ1 job in the *hlq*.SDSNSAMP data set. We also ran sample job DNSTEJ7 for use in the LOB samples.

5.1.1 Installing DB2 SQLJ/JDBC support

We assume that DB2 has been installed correctly, including the JDBC/SQLJ drivers. The JDBC/SQLJ drivers are installed through SMP/E as a separate FMID, called JDB7712 (for DB2 Version 7). The SMP/E installation steps are described in Chapter 6 of *Application Programming Guide and Reference for Java*, SG26-9932-**03**.

As the Java area is a very dynamic area, please make sure you have the latest version of this publication. It is available from the Web at:

http://www-3.ibm.com/software/data/db2/os390/v7books.html

5.1.2 Installing the Universal Driver on a z/OS or OS/390 platform

At the time of writing of this publication, the Universal Driver was not generally available on the z/OS and OS/390 platform. On the z/OS platform, the Universal Driver will be part of the DB2 for z/OS V8 base code. As this functionality is critical for DB2 people deploying Java applications on z/OS, the Universal Driver for SQLJ and JDBC will also be made available for DB2 for z/OS and OS/390 Version 7 (not Version 6) through the maintenance stream.

However, we did some testing using the Type 4 JCC driver on DB2 for z/OS and OS/390. As the Type 4 driver is a pure Java driver, it is sufficient to copy the required jar files into your OS/390 HFS, and set up the classpath properly. We FTPed db2jcc.jar from the directory on the workstation where DB2 for LUW had installed it (...SQLLIB\java) to the classes subdirectory of the DB2 installation directory on USS, /pp/db2javadriver/classes.

5.1.3 Required DB2 for z/OS changes to enable the Universal Driver

To enable certain database meta data catalog queries and message formatting methods when connected to DB2 UDB OS/390 and z/OS, the Universal Driver for SQLJ and JDBC requires certain stored procedures on the target DB2 subsystem.

Required service

The following service is required to enable DB2 V6 and V7 for OS/390 and z/OS for the JCC driver. DB2 V8, when it becomes available, will include this maintenance in the base code.

- PQ62695

  Adds stored procedures, as well as tables and views used by these stored procedures, that allow the JDBC (and ODBC) drivers to retrieve schema-based catalog metadata. As part of installing this maintenance, you need to run the DSNTIJMS job, after customizing it, that lives in the *hlq*.SDSNSAMP library.

  This APAR introduces 13 stored procedures. Twelve of these procedures provide the ability to generate a result set that corresponds to the schema metadata APIs. An additional procedure introduced by this APAR formats SQLCODE message text, given input fields from a DB2-generated SQLCA.

  – SYSIBM.SQLCOLPRIVILEGES
  – SYSIBM.SQLCOLUMNS

  – SYSIBM.SQLFOREIGNKEYS
  – SYSIBM.SQLPRIMARYKEYS
  – SYSIBM.SQLPROCEDURECOLS
  – SYSIBM.SQLPROCEDURES
  – SYSIBM.SQLSPECIALCOLUMNS
  – SYSIBM.SQLSTATISTICS
  – SYSIBM.SQLTABLEPRIVILEGES
  – SYSIBM.SQLTABLES
  – SYSIBM.SQLGETTYPEINFO
  – SYSIBM.SQLUDTS
  – SYSIBM.SQLCAMESSAGE

- PQ72453

  This APAR resolves some errors in the initial implementation of the metadata catalog queries delivered through APAR PQ62695.

> **Note:** You also need to have this support when using the Universal Driver running on a Windows or AIX platform, and you are connecting to the mainframe.

DSNTIJUZ

The *DB2 UDB for OS/390 and z/OS Installation Guide Version 7*, GC26-9936*,* clearly states that you must rebind affected packages after the value has been set to YES.

> **Important:** Make sure the DSNZPARM parameter DESCSTAT is set to YES (Installation Panel DSNTIPF, "Describe for Static" *before* you run the next job. Otherwise, you have to rerun them in order to rebind the affected packages after you make this change.
>
> Do as we say, not as we did.
>
> This option controls whether DB2 builds a DESCRIBE SQLDA when binding static SQL statements. YES means that DB2 does generate a DESCRIBE SQLDA at BIND time so that DESCRIBE requests for static SQL can be satisfied during execution. Specifying YES increases the size of some packages because the DESCRIBE SQLDA is now stored with each statically bound SQL SELECT statement.
>
> In addition, if you use named iterators in your SQLJ programs, and you do *not* use online checking, DESCRIBE FOR STATIC must be set to YES.

This DSNZPARM value may be changed online by issuing the following command:

```
-SET SYSPARM LOAD(dsnzparm_name)
```

DSNTIJMS

This post-install job creates objects required for the DB2 JDBC and ODBC metadata methods. It is straightforward to customize and run. As we want to show you how to set up a WLM application environment later on in this chapter (see "Workload Manager (WLM)" on page 61), we change the WLM ENVIRONMENT parameter on the CREATE PROCEDURE STATEMENTS to DB7YJCC.

T4 Universal Driver setup

When you are directing SQL requests to a DB2 for z/OS and OS/390 from a T4 driver, you need to have some packages bound on the DB2 for z/OS system. This can either be done when setting up a database connection (DB2 Connect) using the DB2 for LUW configuration wizard, or by invoking the DB2Binder utility, for example, from a Windows command prompt, provided your Windows environment is set up properly (see “DB2 Universal Driver - Setup for a Windows environment” on page 71 for details). The command shown in Example 5-1 binds the JDBC packages into a collection called NULLID, the default.

Example 5-1 Invoking the DB2Binder utility

```
java com.ibm.db2.jcc.DB2Binder -url jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y -user bart
-password b1ngo
Bind to "jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y" under collection "NULLID":
Package  "SYSSTAT": Bind succeeded
Package "SYSLH100": Bind succeeded
Package "SYSLH200": Bind succeeded
Package "SYSLH300": Bind succeeded
Package "SYSLH400": Bind succeeded
Package "SYSLN100": Bind succeeded
Package "SYSLN200": Bind succeeded
Package "SYSLN300": Bind succeeded
Package "SYSLN400": Bind succeeded
Package "SYSLH101": Bind succeeded
Package "SYSLH201": Bind succeeded
Package "SYSLH301": Bind succeeded
Package "SYSLH401": Bind succeeded
Package "SYSLN101": Bind succeeded
Package "SYSLN201": Bind succeeded
Package "SYSLN301": Bind succeeded
Package "SYSLN401": Bind succeeded
Package "SYSLH102": Bind succeeded
Package "SYSLH202": Bind succeeded
Package "SYSLH302": Bind succeeded
Package "SYSLH402": Bind succeeded
Package "SYSLN102": Bind succeeded
Package "SYSLN202": Bind succeeded
Package "SYSLN302": Bind succeeded
Package "SYSLN402": Bind succeeded
DB2Binder finished.
```

T2 Universal Driver setup

In this section we discuss T2 Universal Driver setup.

Driver properties

When using the T2 driver, you are using a local (RRS) connection to the DB2 system. Therefore, you need to tell the JDBC driver:

- What DB2 the driver initially has to connect to. You may think that you already indicate this in the URL, when calling the DriverManager or DataSource. However, this is different. This parameter is to tell the JDBC driver to what DB2 the initial connection should be made, irrespective of the name specified in the URL. The db2.jcc.ssid property is used for this purpose. If this property is not specified, the driver will look in the DSNHDECP module for a default DB2 subsystem name to connect to. This may or may not be what you want.
- What plan to execute. Since this is a locally attached driver, we need to tell the driver what plan name to use. When using the T4 driver, we use the DISTSERV plan that is ‘built into’ DB2.

When using the Universal Driver, you can specify these properties in the db2jcct2.properties file. You need to tell the system where this file is by using the export command, like for any other USS environment variable (see Example 5-6 on page 71).

Example 5-2 db2jcct2.properties file

```
db2.jcc.ssid=DB7Y
db2.jcct2.planName=BARTSQLJ
```

Important: At the time of writing of this publication, db2jcct2.properties is the name of the properties file to specify the Type 2 Universal Driver for z/OS properties. This may have changed in the meantime, as the tests were done with a pre-GA version of the driver.

Bind plan

As mentioned before, when you use the Type 2 driver, you need to have a local plan to execute. This applies to both SQLJ and JDBC.

As you can see, we use our own plan, BARTSQLJ, when using the local Type 2 driver. It is bound as shown in Example 5-3. It specifies two collections:

- SG246435 that will contain all our SQLJ packages
- NULLID that contains all our JDBC packages

Example 5-3 Bind plan

```
BIND PLAN(BARTSQLJ) -
  PKLIST(SG246435.*, NULLID.*) -
  ISOLATION CS
```

Important: It is necessary to include the NULLID collection in the PKLIST of your plan. Otherwise executions will fail. This is because the JCC driver needs to do some initial setup work before it starts executing you static package. To to so, it uses a package in the NULLID collection. Therefore if that package from the NULLID collection is not part of the PKLIST, program execution will fail with a -805 SQLCODE.

As part of the JDBC installation steps documented in *Application Programming Guide and Reference for Java™*, SC26-9932, you also bind a default JDBC plan called DSNJDBC. See *prefix*.SDSNSAMP(DSNTJJCL) job for details. You can also use that plan to run your JDBC and SQLJ programs, as long as the PKLIST contains all collections and/or packages that are needed by the applications.

5.2 Workload Manager (WLM)

As the stored procedures used by the DB2 Universal Driver for SQLJ and JDBC are WLM-managed stored procedures, you have to make sure that your WLM environment is set up correctly; more specifically, the WLM application environments.

The following screens are shown to illustrate what is necessary to set up a Workload Manager Application Environment. In most installations, this work is normally performed for you by the z/OS systems staff, as access to the WLM policy and its ISPF panels is usually restricted.

The example below uses an existing policy in which at least the default DB2 WLM Environment has been already created. We modify the existing policy by copying a valid environment to a new one, in this case WLM100, changing the values as desired, saving it, installing the policy, and activating the policy.

Note: Occasionally you may need to change the WLM environment used by some of the stored procedures following installation of some maintenance. You can do this via the ALTER PROCEDURE SQL statement through SPUFI, or via the DB2 Control Center if that is installed.

From the ISPF panel, we enter WL to bring up the WLM panels (yours is likely different). Press Enter on the WLM startup screen, and the panel shown in Figure 5-1 appears.

```
  File  Help
  ---------------------------------------------------------------------------

  Command ===> ______________________________________________________________


                 _____________________________________________
                |         Choose Service Definition           |
                |                                             |
                | Select one of the following options.        |
                | 2   1.  Read saved definition               |
                |     2.  Extract definition from WLM         |
                |         couple data set                     |
                |     3.  Create new definition               |
                |                                             |
                |                                             |
                |                                             |
                |_____________________________________________|

                              ENTER to continue
```

Figure 5-1   WLM - Choose a service definition

Use option 2 and press Enter to extract the definition from the existing and active WLM policy. Always do this to avoid regressing the existing policy.

Expect this to take a few minutes. The next panel is Figure 5-2 on page 63.

```
File  Utilities  Notes  Options  Help
 --------------------------------------------------------------------------
 Functionality LEVEL004        Definition Menu        WLM Appl LEVEL013
 Command ===> ______________________________________________________________

 Definition data set  . . : none

 Definition name  . . . . . Sampdef   (Required)
 Description  . . . . . . . Sample WLM Service Definition

 Select one of the
 following options. . . . . 9__  1.   Policies
                                 2.   Workloads
                                 3.   Resource Groups
                                 4.   Service Classes
                                 5.   Classification Groups
                                 6.   Classification Rules
                                 7.   Report Classes
                                 8.   Service Coefficients/Options
                                 9.   Application Environments
                                 10.  Scheduling Environments
```

Figure 5-2 WLM - Choose options from the extracted policy

We select option 9 and press Enter to look at the existing application environments. Figure 5-3 lists all existing WLM Application Environments.

```
Application-Environment  Notes  Options  Help
 --------------------------------------------------------------------------
                  Application Environment Selection List     Row 1 to 12 of 12
 Command ===> ______________________________________________________________

 Action Codes: 1=Create, 2=Copy, 3=Modify, 4=Browse, 5=Print, 6=Delete,
               /=Menu Bar

 Action  Application Environment Name      Description
   __    BARTSRV                           J2EE Application Server for Bart
   __    BBOASR1                           WAS IVP Server
   __    BBOASR2                           J2EE Application Server
   __    CBINTFRP                          WAS Interface Repository Server
   __    CBNAMING                          WAS Naming Server
   __    CBSYSMGT                          WAS System Management Server
   __    DB7YJSPP                          db7y DSNTJSPP (SPB)
   __    DB7YREXX                          For Rexx SPs DSNTPSMP/DSNTBIND
   __    DB7YUTIL                          DB7Y stored proc Utility
   2_    WLMENV                            DB7Y all non-spcial SPs
   __    WLMENV1                           DB7Y stored proc environment


 ******************************* Bottom of data ********************************
```

Figure 5-3 WLM - Select an application environment

Since copying is the sincerest form of productivity, choose key 2 beside an existing DB2 WLM Environment and press Enter. We see Figure 5-4.

```
 Application-Environment  Notes  Options  Help
 ---------------------------------------------------------------------------
                      Copy an Application Environment
 Command ===> ______________________________________________________________

 Application Environment  . . . ________________________________  Required
 Description  . . . . . . . . . DB7Y all non-spcial SPs
 Subsystem Type . . . . . . . . DB2
 Procedure Name . . . . . . . . DB7YWLM
 Start Parameters . . . . . . . DB2SSN=DB7Y,NUMTCB=8,APPLENV=WLMENV
                                ________________________________________
                                ___________________________________

 Limit on starting server address spaces for a subsystem instance:
 1   1.  No limit
     2.  Single address space per system
     3.  Single address space per sysplex
```

Figure 5-4   WLM - Copy an application environment

WLMENV uses the default environment we chose on the DB2 Installation Panels at DB2 installation time. This application environment name is used for all stored procedures that ship with DB2, which have no specific requirements. That is, there is no requirement to single thread them (TCB=1), as is required for DSNUTILS (the utility stored procedure) and DSNTPSMP (the SQL Stored Procedure Builder). We could have used WMLENV for our stored procedures as well, but since we want to show you how to set up a WLM application environment, we chose to create our own, just for the stored procedures used by the DB2 Universal Driver for Java Common Connectivity. In Figure 5-5 we make the necessary changes to the environment.

1. Specify the name of the application environment DB7YJCC.

```
 Application-Environment  Notes  Options  Help
 ---------------------------------------------------------------------------
                      Copy an Application Environment  "FORWARD " is not active
 Command ===> ______________________________________________________________

 Application Environment  . . . DB7YJCC                           Required
 Description  . . . . . . . . . DB7Y stored procs used by JCC
 Subsystem Type . . . . . . . . DB2
 Procedure Name . . . . . . . . DB7YWLM
 Start Parameters . . . . . . . DB2SSN=DB7Y,NUMTCB=8,APPLENV=DB7YJCC
                                ________________________________________
                                ___________________________________

 Limit on starting server address spaces for a subsystem instance:
 1   1.  No limit
     2.  Single address space per system
     3.  Single address space per sysplex
```

Figure 5-5   WLM - Create application environment WLM100

2. Provide a meaningful description of the purpose of the environment.

3. Specify APPLENV=DB7YJCC as the same name on the application environment, and DB2SSN=DB7Y as the DB2 subsystem this application environment address space will connect to, in the start parameters.
4. We indicate that there is no limit on the number of address spaces that we allow WLM to start.
5. As we reused the existing DB7YPROC, there is no need to create a new started task by that name in SYS1.PROCLIB. If you want to use a different one, someone may have to do this for you, as access to PROCLIB is usually restricted. Example 5-4 on page 67 shows the cataloged procedure we use.

After entering all values described above, we press Enter. Figure 5-6 appears.

```
Application-Environment  Notes  Options  Help
---------------------------------------------------------------------------
                  Application Environment Selection List     Row 1 to 17 of 21
Command ===> ______________________________________________________________

Action Codes: 1=Create, 2=Copy, 3=Modify, 4=Browse, 5=Print, 6=Delete,
              /=Menu Bar

Action  Application Environment Name      Description
  __    BARTSRV                           J2EE Application Server for Bart
  __    BBOASR1                           WAS IVP Server
  __    BBOASR2                           J2EE Application Server
  __    CBINTFRP                          WAS Interface Repository Server
  __    CBNAMING                          WAS Naming Server
  __    CBSYSMGT                          WAS System Management Server
  __    DB7YJCC                           DB7Y all JCC  SPs
  __    DB7YJSPP                          db7y DSNTJSPP (SPB)
  __    DB7YREXX                          For Rexx SPs DSNTPSMP/DSNTBIND
  __    DB7YSPB2                          DSNTPSMP proc for V1.15 update
  __    DB7YSQL                           Execution proc for SQL SPs
  __    DB7YUTIL                          DB7Y stored proc Utility
  __    PKINTFRP                          PKI WAS Interface Repo Server
  __    PKNAMING                          PKI WAS Naming Server
```

Figure 5-6 DB7YJCC environment now created

The Application Environment Selection List is displayed, showing our new environment, DB7YJCC.

Now we click **Save** (PF3 normally) multiple times until we get back to the panel displayed in Figure 5-7 on page 66.

```
File  Utilities  Notes  Options  Help
--------------------------------------------------------------------------
Functionality LEVEL011        Definition Menu        WLM Appl LEVEL013
Command ===> ______________________________________________________________

Definition data set  . . : none
               ______________________________________________________________
Definition na |         Specify disposition of Service Definition           |
Description   |                                                             |
              | The service definition has been changed but not saved.      |
Select one of |                                                             |
following opt | Select one of the following options or PF12 to go back to   |
              | the WLM Administrative Application.                         |
              | 2_  1.  Save definition to data set and continue            |
              |     2.  Install definition on WLM couple data               |
              |         set and continue                                    |
              |     3.  Discard changes and continue                        |
              |                                                             |
              |                                                             |
              |_____________________________________________________________|
                                10.  Scheduling Environments
```

Figure 5-7 WLM - Install the WLM definition

Specify option 2 and press Enter again. This takes you back to the WLM primary menu.

From the Utilities drop-down menu, we select option 3 to activate the policy, as shown in Figure 5-8.

```
File  Utilities  Notes  Options  Help
 -----  ________________________________________________  ----------------
 Funct |  3 1. Install definition                          |  Appl LEVEL013
 Comma |    2. Extract definition                          | ________________
       |    3. Activate service policy                     |
 Defin |    4. Allocate couple data set                    |
       |    5. Allocate couple data set using CDS values   |
 Defin |    6. Validate definition                         |
 Descr |___________________________________________________|

 Select one of the
 following options. . . . . ___  1.   Policies
                                 2.   Workloads
                                 3.   Resource Groups
                                 4.   Service Classes
                                 5.   Classification Groups
                                 6.   Classification Rules
                                 7.   Report Classes
                                 8.   Service Coefficients/Options
                                 9.   Application Environments
                                 10.  Scheduling Environments
```

Figure 5-8 WLM utilities - Choose to activate the policy

Figure 5-9 on page 67 lists the services policies currently installed in the WLM Couple Data Sets.

```
File  Utilities  Notes  Options  Help
 -  _______________________________________________________________________
 F |                        Policy Selection List         Row 1 to 1 of 1 |
 C | Command ===> ________________________________________________________ |
   |                                                                      |
 D |The following is the current Service Definition installed on the WLM  |
   |couple data set.                                                      |
 D |                                                                      |
 D | Name . . . . : Sampdef                                               |
   |                                                                      |
 S | Installed by : BARTR2    from system  SC63                           |
 f | Installed on : 2002/10/16  at  20:33:49                              |
   |                                                                      |
   | Select the policy to be activated with "/"                           |
   |                                                                      |
   | Sel  Name      Description                                           |
   |  /   WLMPOL    Sample WLM Service Policy                             |
   | ************************** Bottom of data *************************** |
   |                                                                      |
 - |______________________________________________________________________|
```

Figure 5-9 Activate the modified WLM policy

Select WLMPOL by specifying a / (slash) beside the service policy and press Enter.

When the screen is redisplayed, you should see the words `Service policy WLMPOL was activated.` (`IWMAM060`) at the bottom.

Verify that the application environment DB7YJCC is available by issuing the following z/OS console command:

```
/D WLM,APPLENV=*
```

If you specified a new procedure, make sure to add it to a member of a data set in the PROCLIB concatenation of your system. See Example 5-4 below.

Example 5-4 Started task JCL for DB7YWLM in SYS1.PROCLIB

```
//DB7YWLM PROC RGN=OK,APPLENV=XXXXXX,DB2SSN=DB7Y,NUMTCB=8
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
//        PARM='&DB2SSN,&NUMTCB,&APPLENV'
//STEPLIB  DD  DISP=SHR,DSN=DB7YU.RUNLIB.LOAD
//         DD  DISP=SHR,DSN=CEE.SCEERUN
//         DD  DISP=SHR,DSN=DB7Y7.SDSNLOAD
//DSSPRINT DD  SYSOUT=*
//SYSIN    DD  UNIT=SYSDA,SPACE=(4000,(20,20),,,ROUND)
//SYSPRINT DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

Note that the APPLENV parameter in the procedure is the default WMLENV. The parameter is overwritten at address space startup time with DB7YJCC, as specified in the WLM application environment startup parameter.

5.3 Unix System Services

There are many different ways to run a Java program under Unix System Services (USS) on a z/OS or OS/390 system. You can run it as a batch job, from a command line, under the control of WAS, etcetera. In this section we show how to set up your USS environment to be able to run a sample Java program from the command prompt in USS.

5.3.1 Setting up a USS session

To illustrate how to run our sample program under a USS command prompt, we first have to establish a Unix session. Here we chose to use a Telnet session to log onto USS using IBM Personal Communications (PCOM). If you use another tool, the setup may look different from ours, shown in Figure 5-10.

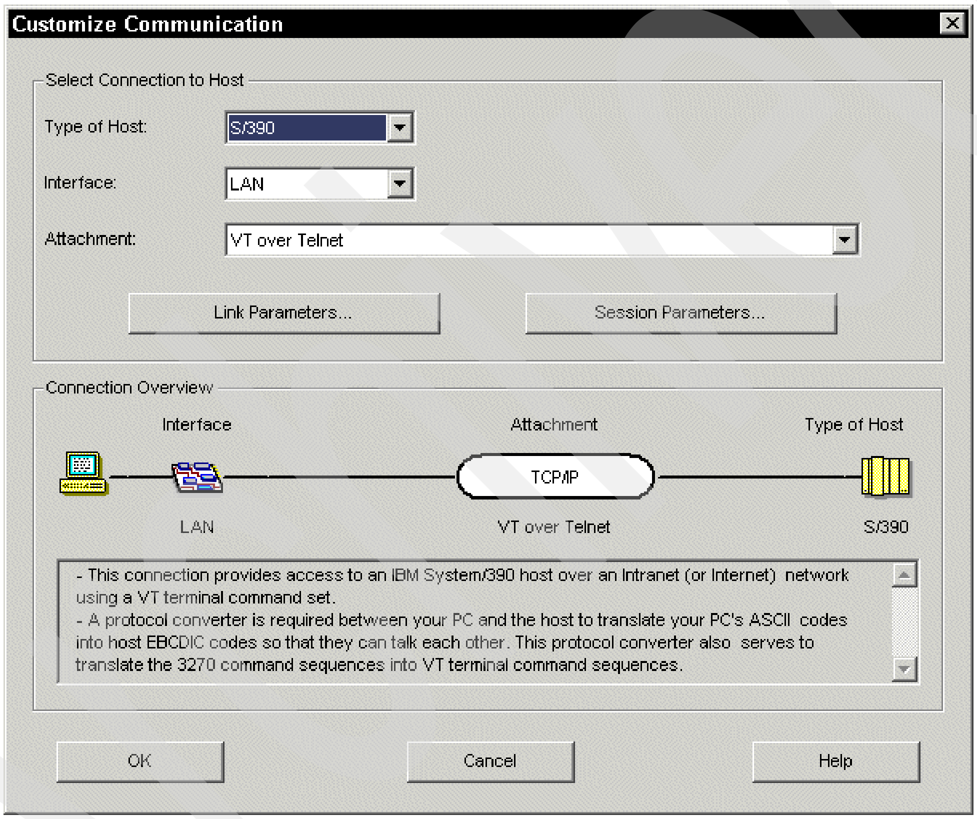

Figure 5-10 PCOM Telnet screen after choosing Communication -> Configure -> Attachment

Change the Attachment to "VT over Telnet".

Then click the **Link Parameters** button to produce the screen shown in Figure 5-11 on page 69.

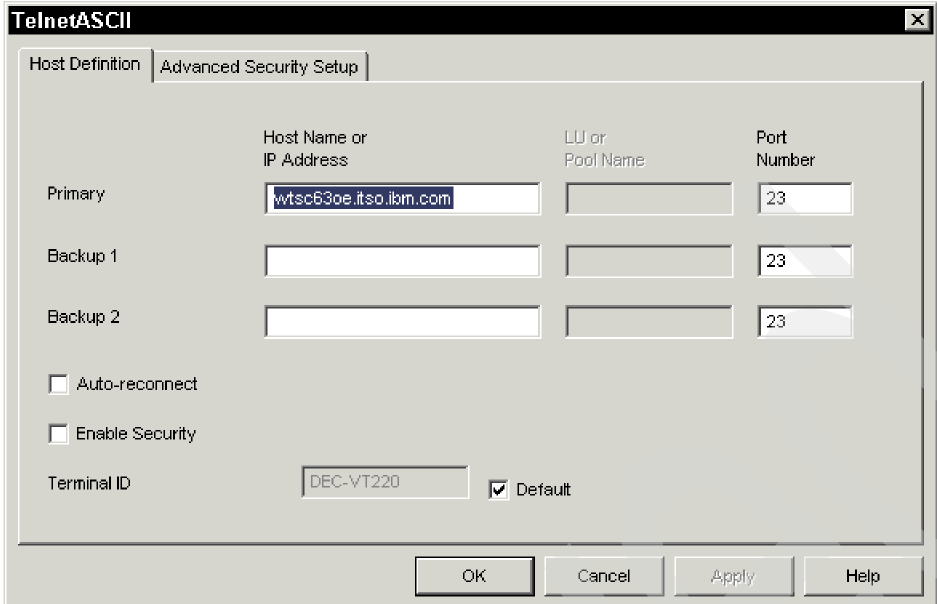

Figure 5-11 Link Parameters changed - Adding the host DNS/port

Specify the host name (or IP address). Our system had been configured to use native USS using the host name WTSC63OE. It is also configured to use port number 23. Find out what your values are. Then click **OK** to return to the Customize Communication screen.

Click the **Session Parameters** button and go to Figure 5-12.

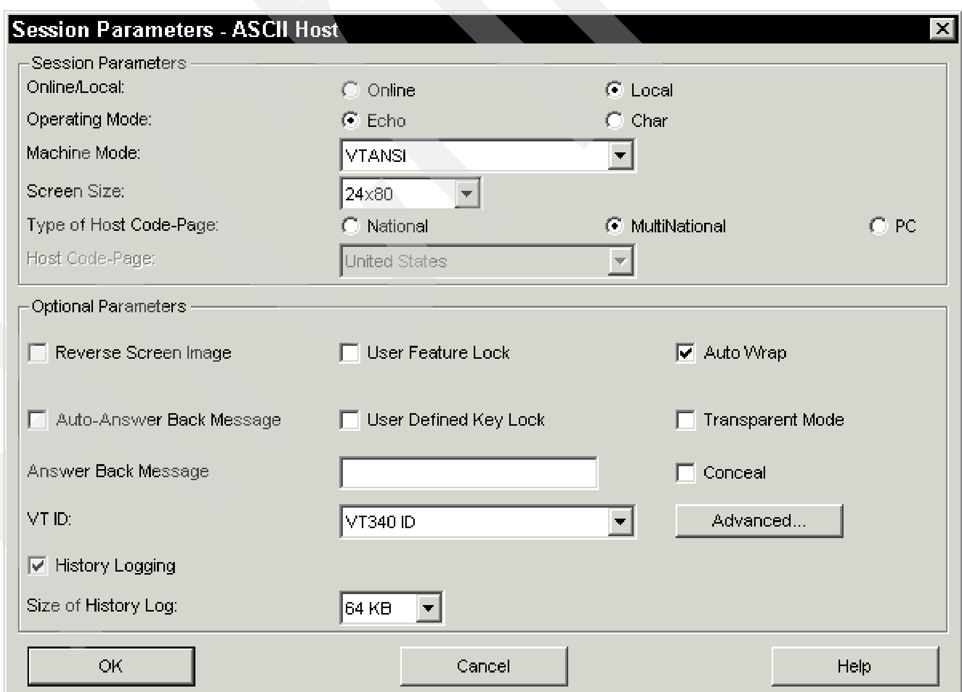

Figure 5-12 Session parameters changed

We chose to use VTANSI for machine mode, and checked Autowrap.

Click **OK**, exit the panels, and then save the configuration using the File toolbar and select **Save As**. Give the session a name. You are also prompted whether or not to create an icon on the desktop, so you can easily log onto USS.

5.3.2 Setting up the JDBC/SQLJ environment variables

In order for your Java environment to work properly under USS, you need to make sure that you have set up your environment variables correctly.

The STEPLIB environment variable on Unix System Services

The data sets SDSNEXIT, SDSNLOAD and SDSNLOD2 must be included in the STEPLIB environment variable. The form of this environment variable is different than the norm, in that it names these data sets using the MVS file system naming convention. For example, the name used could be DB7Y7.SDSNLOAD, where DB7Y7 is the high-level qualifier for the DB2 libraries, for example:

```
DB7Y7.SDSNEXIT:DB7Y7.SDSNLOAD:DB7Y7.SDSNLOD2
```

This environment variable does not need to be set if these data sets are present in the linklist.

The PATH environment variable

This directory that contains the commands or shell scripts that invoke Java, JDBC, and SQLJ program preparation and debugging functions, such as `javac`, `java`, `db2sqljcustomize`, must be installed in the PATH environment variable. Under Unix System Services these are commonly in the directory:

```
/pp/db2javadriver/bin
```

The LIBPATH and LD_LIBRARY_PATH environment variables

Dynamic load libraries (DLLs) contain the native code portion of Type 2 drivers. If you are using such a driver, then the directory containing these DLLs must be added to the LIBPATH and LD_LIBRARY_PATH environment variables. Under Unix System Services these DLLs are commonly in the directory:

```
/pp/db2javadriver/lib
```

The CLASSPATH environment variable

This is probably the most important environment variable when using JDBC (and Java in general). This environment variable points to the specific class file(s) for the JDBC driver you are using in your application. This entry needs to match the driver that you are using in your application. To see the relation between the drivers and the class files, refer to Table 3-1 on page 26.

The CLASSPATH can be coded directly with the **`java`** command, when running a Java application from the command line, or from within a shell script using the -cp option. Example 5-5 shows how to set the CLASSPATH when executing a Java program from the command line (it assumes that the variable DB2HOME has been set to your DB2 installation directory, usually /usr/lpp/db2/db2710. However, we used /pp/db2javadriver as our home directory).

Example 5-5 Setting the CLASSPATH from the command line in Unix System Services

```
java -cp $DB2HOME/classes/db2jcc.jar:$DB2HOME/classes/db2jcc_license_cisuz.jar
 com/ibm/itso/testJDBC/TestJDBC.class
```

Setting environment variables

These variables can be set by hand for each session, but it is more common to include the settings in the .profile file in one's home directory. Under Unix System Services the home directory is set up in the RACF® profile for the user. A USS profile is analogous to a TSO logon clist. It includes the libraries and other settings that the USS user needs to perform his activities.

The **export** command is used to set up the environment variable. For example, to put the file db2jcc.jar in your CLASSPATH, you can specify the following command, either on the command line or by placing it in your .profile:

```
export CLASSPATH=/pp/db2javadriver/classes/db2jcc.jar:$CLASSPATH
```

The $CLASSPATH entry represents the current value in CLASSPATH, so this command adds the jar file to the beginning of this environment variable.

Files that are added to these variables must be fully qualified, so the example assumes that the required file is in the /usr/lpp/db2/db2/db2710/classes directory. (The default installation directory is /usr/lpp/db2/db2710. Because we use a test version of the new JCC driver on the same system that has the old JDBC/SQLJ for z/OSDB2 driver, we decided to install the JCC driver in a different directory, namely /pp/db2javadriver.)

Example 5-6 shows our profile. With these settings we managed to run the sample Java applications in this publication.

Example 5-6 USS - User profile

```
# This line exports the variable settings so that they are known to the
# system.
export PATH EDITOR PS1
DB2HOME=/pp/db2javadriver
CLASSPATH=$DB2HOME/classes/sqlj.zip:$CLASSPATH
CLASSPATH=$DB2HOME/classes/db2jcc.jar:$CLASSPATH
CLASSPATH=$DB2HOME/classes/db2jcc_license_cisuz.jar:$CLASSPATH
CLASSPATH=$DB2HOME/classes/db2jcc_javax.jar:.:$CLASSPATH
export CLASSPATH
export JAVA_HOME=/usr/lpp/java/IBM/J1.3
export LIBPATH=$DB2HOME/lib:$LIBPATH
LD_LIBRARY_PATH=$DB2HOME/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
PATH=$DB2HOME/bin:$PATH
export PATH=$JAVA_HOME/bin:$PATH
export STEPLIB=DB7Y7.SDSNEXIT:DB7Y7.SDSNLOAD:DB7Y7.SDSNLOD2
export DB2JCCPROPERTIES=/u/bart/db2jcct2.properties
```

When you run the samples that use the DB2 Universal Driver and still have the legacy drivers (JDBC/SQLJ driver for z/OS) in your CLASSPATH, you must make sure that the db2jcc.jar file is in the CLASSPATH, ahead of the legacy driver files, because some classes have the same name in the legacy and the DB2 Universal Driver for SQLJ and JDBC. However, to avoid confusion it is probably best to only have one set of drivers (old or new) in the CLASSPATH at any one time.

5.4 DB2 Universal Driver - Setup for a Windows environment

DB2 UDB for Linux, Unix and Windows JDBC support is provided as part of the Java Enablement option on DB2 clients and servers. With this support, you can build and run

JDBC applications and applets. These contain dynamic SQL only, and use a Java call interface to pass SQL statements to DB2.

DB2 Java embedded SQL (SQLJ) support is provided as part of the DB2 AD Client. With DB2 SQLJ support, in addition to DB2 JDBC support, you can build and run SQLJ applets and applications. These contain static SQL and use embedded SQL statements that are bound to a DB2 database.

The SQLJ support provided by the DB2 AD Client includes:

- The *DB2 SQLJ translator*, sqlj, which replaces embedded SQL statements in the SQLJ program with Java source statements, and generates a serialized profile that contains information about the SQL operations found in the SQLJ program
- The *DB2 SQLJ profile customizer*, db2sqljcustomize, which precompiles the SQL statements stored in the serialized profile, customizes them into runtime function calls, and by default generates a package in the DB2 database
- The *DB2 SQLJ profile binder*, db2sqljbind, which generates packages from a previously customized SQLJ program
- The *DB2 SQLJ profile printer*, db2sqljprint, which prints the contents of a DB2 customized version of a profile in plain text

The IBM DB2 Universal Driver for SQLJ and JDBC was first shipped when DB2 for Linux, Unix and Windows Version 8 became generally available. At that time only the Type 4 JDBC driver was available. DB2 UDB for LUW FixPak 2 added the Type 2 driver to the Universal Driver family.

Most of our samples throughout this publication use WebSphere Studio, running on a Windows environment, but you can also run Java programs natively from a command prompt.

The Universal Driver for SQLJ and JDBC is installed as part of the normal DB2 UDB for LUW V8 installation process. The default location of the driver is c:\Program Files\IBM\SQLLIB\java. However, since most people tend to install DB2 in c:\SQLLIB, we have tried to consistently use that directory throughout this publication.

Updating the environment variables

The DB2 installation process normally takes care of updating the CLASSPATH environment variable, but it does not hurt to check again to make sure. In addition, when you want to run Java applications from the command line (**Start -> Programs -> DB2 -> Command Line Tools -> Command WIndow**), you have to make sure the PATH environment variable is set up correctly. Also, when running the Ant scripts in WSAD to do SQLJ customization (see “Preparing SQLJ programs to use static SQL through WSAD” on page 157), they use the Windows classpath settings.

To check the environment variables on a Windows system, you can select **Start -> Control Panel -> System -> Advanced -> Environment Variables**.

In the System Variables pane, highlight the environment variable that you want to edit and click **Edit System Variable**.

PATH Make sure to include a supported Java SDK in your PATH environment variable, for example, C:\SQLLIB\java\jdk\bin, if you want to run Java programs from the command line. The DB2 installation process does not do this. If you want to use SQLJ, and need to invoke the SQLJ translator (SQLJ.exe), make sure that C:\SQLLIB\bin is in the PATH as well. Normally this should have been taken care of by the DB2 installation process.

CLASSPATH Make sure the following .jar and .zip files are included in the CLASSPATH:

—"." (The current directory)

—The file C:\sqllib\java\db2jcc.jar

—The file C:\sqllib\java\db2jcc_license_cisuz.jar (license file to be able to connect to a DB2 for z/OS and OS/390 using the Type 4 driver)

To build SQLJ programs, the CLASSPATH should also include the file C:\sqllib\java\sqlj.zip.

TCP/IP listening port

To build applications that access DB2 UDB for LUW with the JDBC Universal Type 2 or JDBC Universal Type 4 Driver, or to build applets with the JDBC Universal Type 4 Driver, the TCP/IP listener must be running. To ensure this, do the following from a DB2 command window:

1. Set the environment variable DB2COMM to TCPIP as follows:

   ```
   db2set DB2COMM=TCPIP
   ```

2. Update the database manager configuration file with the TCP/IP service name as specified in the services file. The services file can be found in the C:\WINNT\system32\drivers\etc directory.

   ```
   db2 update dbm cfg using SVCENAME <TCP/IP service name>
   ```

You must do a **db2stop** and **db2start** for this setting to take effect. Note also that the port number used for applets and SQLJ programs needs to be the same as the TCP/IP SVCENAME number used in the database manager configuration file.

As we are focussing on accessing DB2 for z/OS and OS/390 data, this does not really apply to our samples. However, in many installations people develop on a workstation and test against local DB2 data on that workstation. In that case, the listener port has to be set up properly in order for the Universal Driver to work.

5.5 WSAD setup

As mentioned before, we use WebSphere Studio Application Developer (WSAD) for most of our samples in this publication.

Since WSAD V5.1 was not yet available at the time we wrote this publication, we downloaded a pre-release version of WSAD V5.1 from an IBM internal site.

Since your version will likely use a CD, your installation process will be slightly different.

Click the **Setup.exe** in the folder into which you want the unzipped WSAD files. We only need to install WebSphere Studio Application Developer, not Rational Clear Case or the IBM Agent Controller. In the Custom Setup section, we did not modify any of the options. For the version that we use, we install only the “Required Features”, which include the Integrated Development Environment (IDE) and the Runtimes for both WAS Version 4 and WAS Version 5.

We install WSAD on the default directory c:\Program Files\IBM\WebSphere Studio.

To start WSAD, select **Start -> IBM WebSphere Studio -> Application Developer**.

The first screen to pop-up should look like Figure 5-13 on page 74.

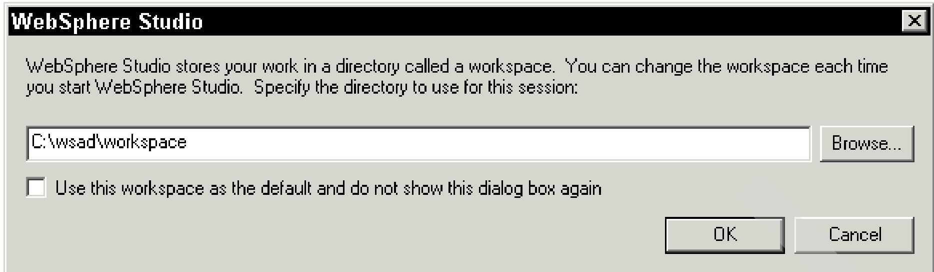

Figure 5-13 Tell WSAD where to put its workspace

We direct WSAD to save our work in C:\wsad\workspace. The default is c:\Documents and Settings. During some of the sample executions, we need to specify the location of the workspace.

After the WSAD logo screen, and being patient, we get into the actual WSAD workbench with a Welcome to WebSphere Studio panel.

To make sure we installed WSAD correctly and did a good DB2 and JCC setup, we test the setup by using some of the functions of the WSAD Data Perspective.

5.5.1 Using the data perspective

Choose the Data Perspective (**Window -> Open Perspective -> Other... -> Data**). A screen like Figure 5-14 on page 75 with the data perspective normally appears.

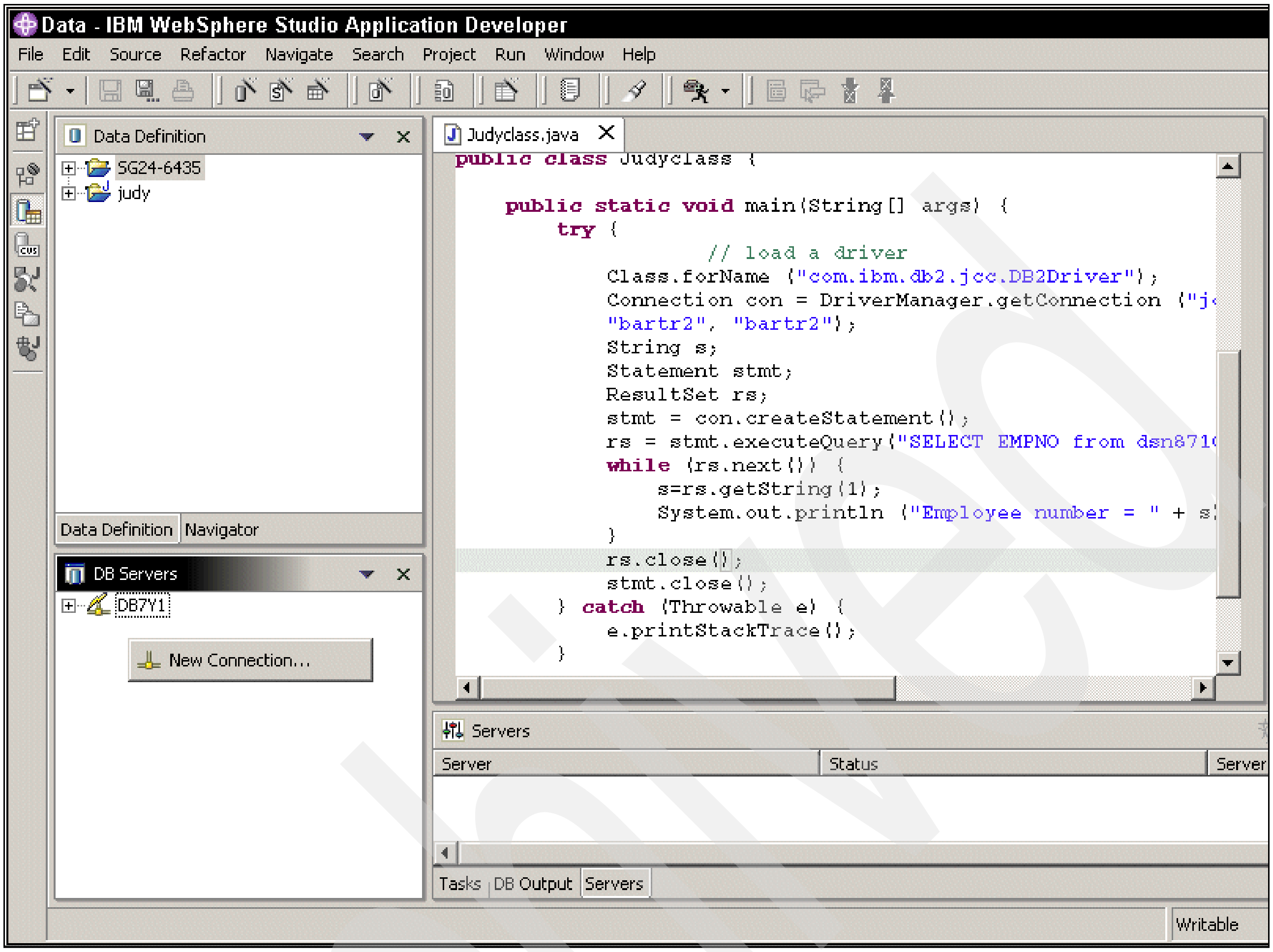

Figure 5-14 WSAD - The data perspective

The DB Servers pane is located on the lower left-hand side of the perspective. (If you have never created a connection to a DB Server, this pane is blank.) As we want to create a connection to allow us to communicate with our DB2 for z/OS and OS/390 (V7) subsystem, click anywhere in the white part of the DB Servers pane, and the words `New Connection` appear. Click the **New Connection** box, and Figure 5-15 on page 76 appears.

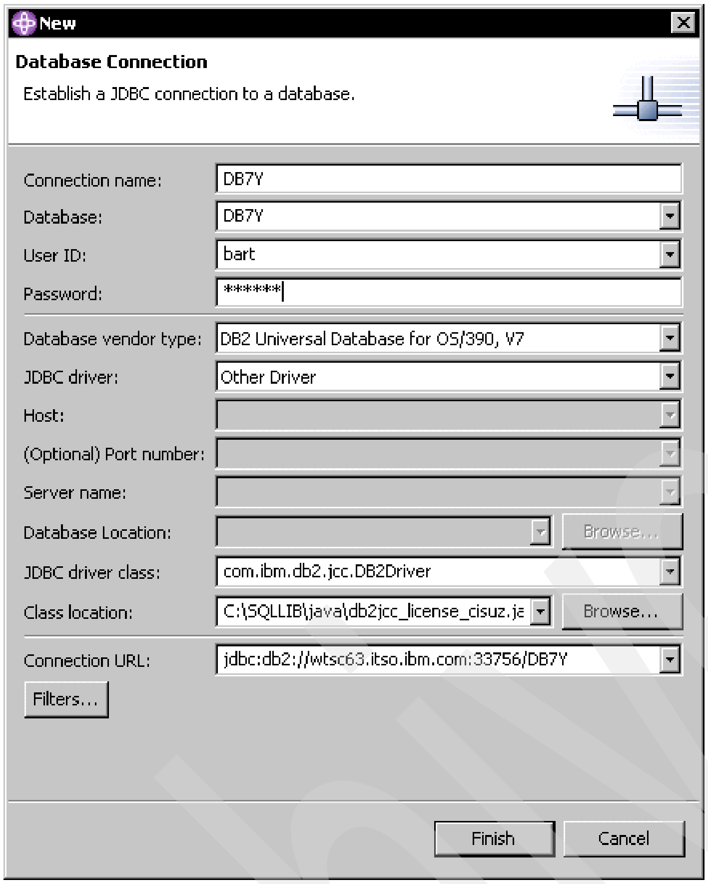

Figure 5-15 Creating a new connection using the DB2 UDB V8 Type 4 Universal Driver

We have to change many things on this panel. We need the information from the DSNL004I message issued by DDF as it starts. It can be found in the job log of the DB2 master address space, DB7YMSTR in our configuration, as shown in Example 5-7. You can also use the output of the -DISPLAY DDF command if your DB2 is Version 7 or later.

- Connection name: You can specify any descriptive string as the connection name.

Example 5-7 DSNL004I - DB2/390 DDF information used for Control Center and WSAD

```
DSNL004I  -DB7Y DDF START COMPLETE  867
            LOCATION  DB7Y
            LU        USIBMSC.SCDB7Y
            GENERICLU -NONE
            DOMAIN    wtsc63.itso.ibm.com
            TCPPORT   33756
            RESPORT   33757
 DSN9022I  -DB7Y DSNYASCP 'STA DB2' NORMAL COMPLETION
```

- Database: You must specify the DB2 location name here.
- User ID and password for our DB2 for OS/390 V7 system DB7Y.
- Database vendor type: Select **DB2 Universal Database for OS/390 , V7** from the drop-down list.
- JDBC driver: We choose **Other DB2/390 Driver**. It un-greys the driver class and location fields.

- For JDBC driver class, we coded the name "com.ibm.db2.jcc.DB2Driver" as the name for the DB2 UDB V8 Universal Driver (Type 4).
- We use **Browse** to find the class location. It is in the directory where we installed DB2 UDB V8.1 (we used C:\SQLLIB\Java, but the default is C:\Program Files\IBM\SQLLIB\Java). Locate the correct Universal Driver, db2jcc.jar, in the Java subdirectory (Figure 5-16), as well as the appropriate licence jar file (db2jcc_license_cisuz.jar because we are going to a DB2 for z/OS and OS/390 system). The license jar is provided with the DB2 Connect product, so in order to be able to use the Type 4 driver, you must have a DB2 Connect license.

> **Important:** As of DB2 UDB V8.1.2, the DB2 Universal Driver for SQLJ and JDBC requires a license jar file. It has to be in the CLASSPATH along with the db2jcc.jar file. Here are the required license JAR files:
>
> - db2jcc_license_c.jar
>
>   Permits JDBC connectivity to DB2J (that is, Cloudscape) servers only. Cloudscape is bundled with WAS along with this license.
>
> - db2jcc_license_cu.jar
>
>   Permits JDBC/SQLJ connectivity to all DB2 LUW (Linux, Unix, Windows) servers and Cloudscape. This is the standard license provided with UDB on Unix/Windows. db2jcc_license_cu.jar is included with all of the DB2 LUW products, including Personal Edition (PE), Websphere Edition (WSE), and DB2 Express.
>
> - db2jcc_license_cisuz.jar
>
>   Permits JDBC/SQLJ connectivity to all DB2 servers, including z/OS, iSeries, VM/VSE DB2 products, UDB for Unix/Windows, and Cloudscape. This license is provided to DB2 Connect licensees only. db2jcc_license_cisuz.jar is included with all DB2 Connect products including Personal Edition, Enterprise Edition (CUE/CASE), and DB2 ESE.
>
> The meaning of the suffix letters in the license file names is as follows: c=cloudscape, i=iSeries, z=z/OS, s=sqlds, u=unix/windows.

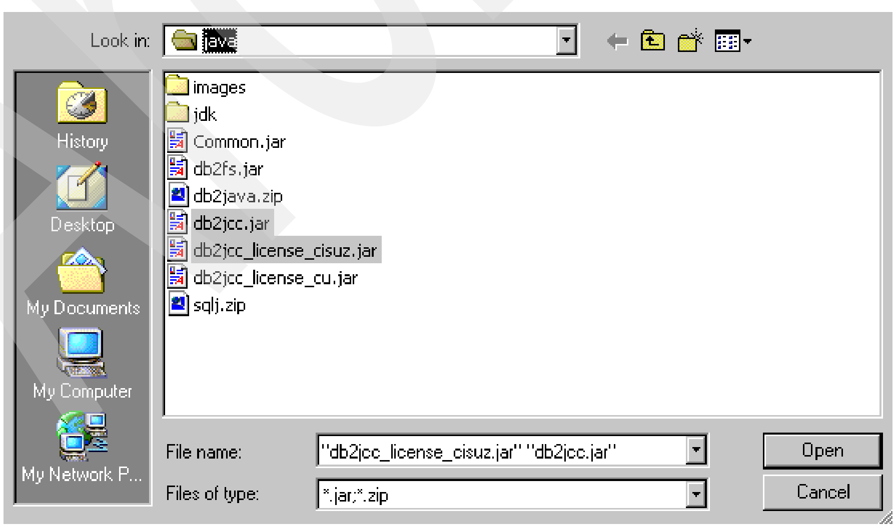

Figure 5-16 Select jar files

- The last entry was the connection URL. It is specified as `jdbc:db2://`*`server`*`:`*`port`*`/`*`Location_name`*. This form of invocation tells the JDBC driver that we want a Type 4 connection. We coded our values. If DDF in your installation is listening on well known port “446”, you do not have to enter the port number.

When pressing the **Finish** button, you are kindly reminded that a DB2 for z/OS and OS/390 subsystem can contain a large number of objects, and that it may be appropriate to set up a filter (Figure 5-17).

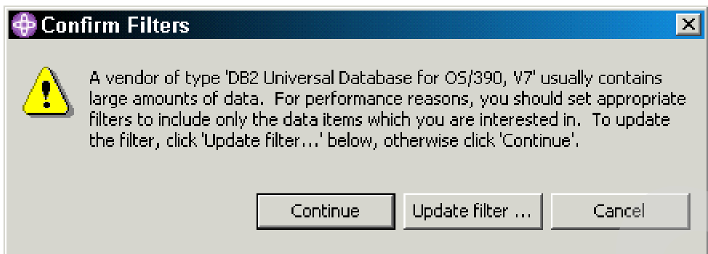

Figure 5-17 Confirm Filters

As our DB2 system is very small, we click **Continue**. Figure 5-15 on page 76 appears again, with the indication at the bottom `Establishing connection to ‘DB7Y’`. If everything is ok, Figure 5-19 on page 79 appears. Otherwise you receive an error message like Figure 5-18. In this case, the network is down, and we cannot get to the host system that is running our DB7Y subsystem.

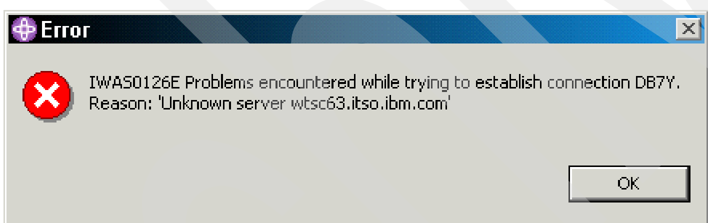

Figure 5-18 Database connection error

In the lower left hand side, DB7Y was added to DB2 Servers.

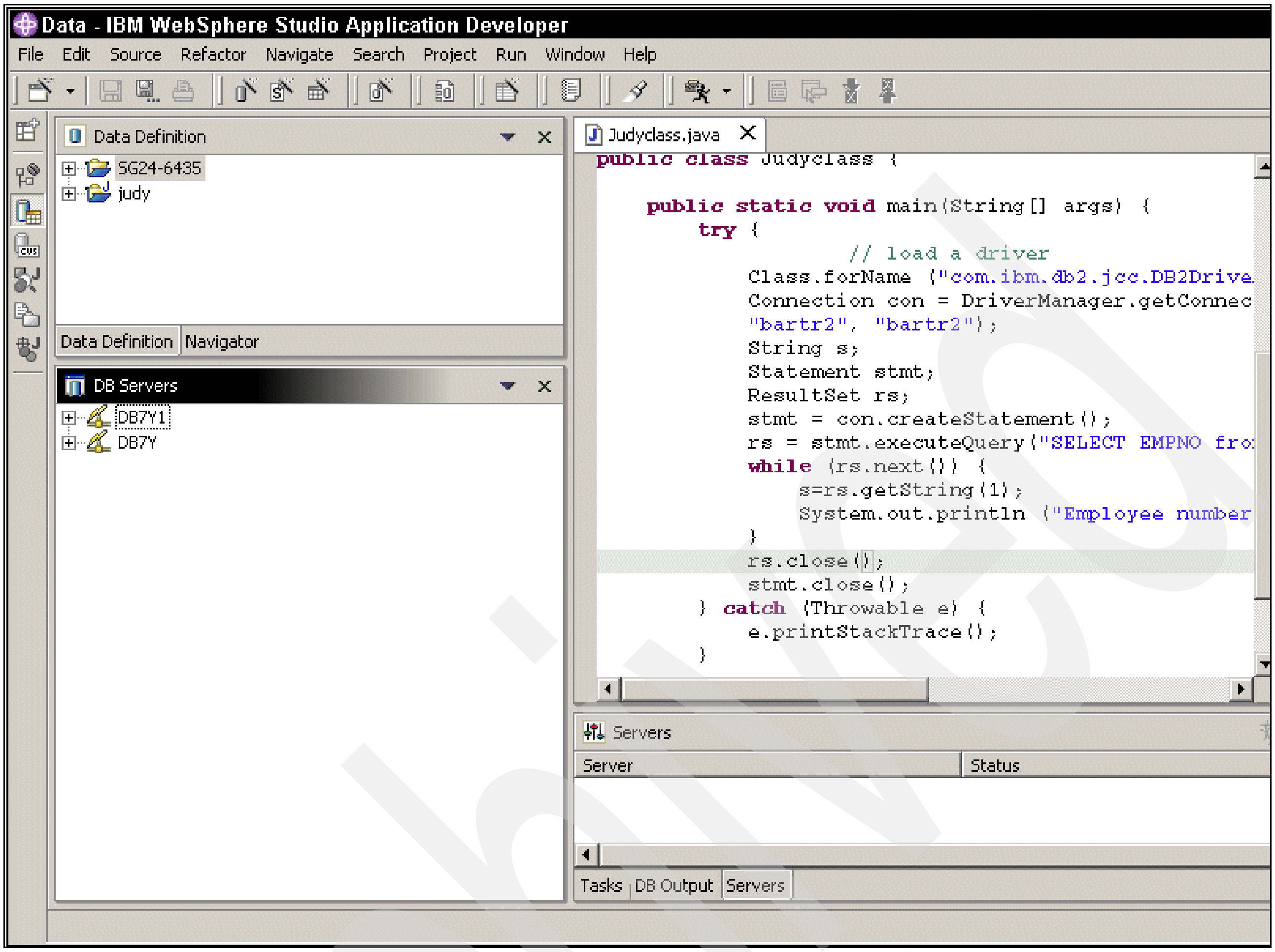

Figure 5-19 The completed connection DB7Y

Next, we expand the DB Server DB7Y by clicking the **+** sign (Figure 5-20 on page 80).

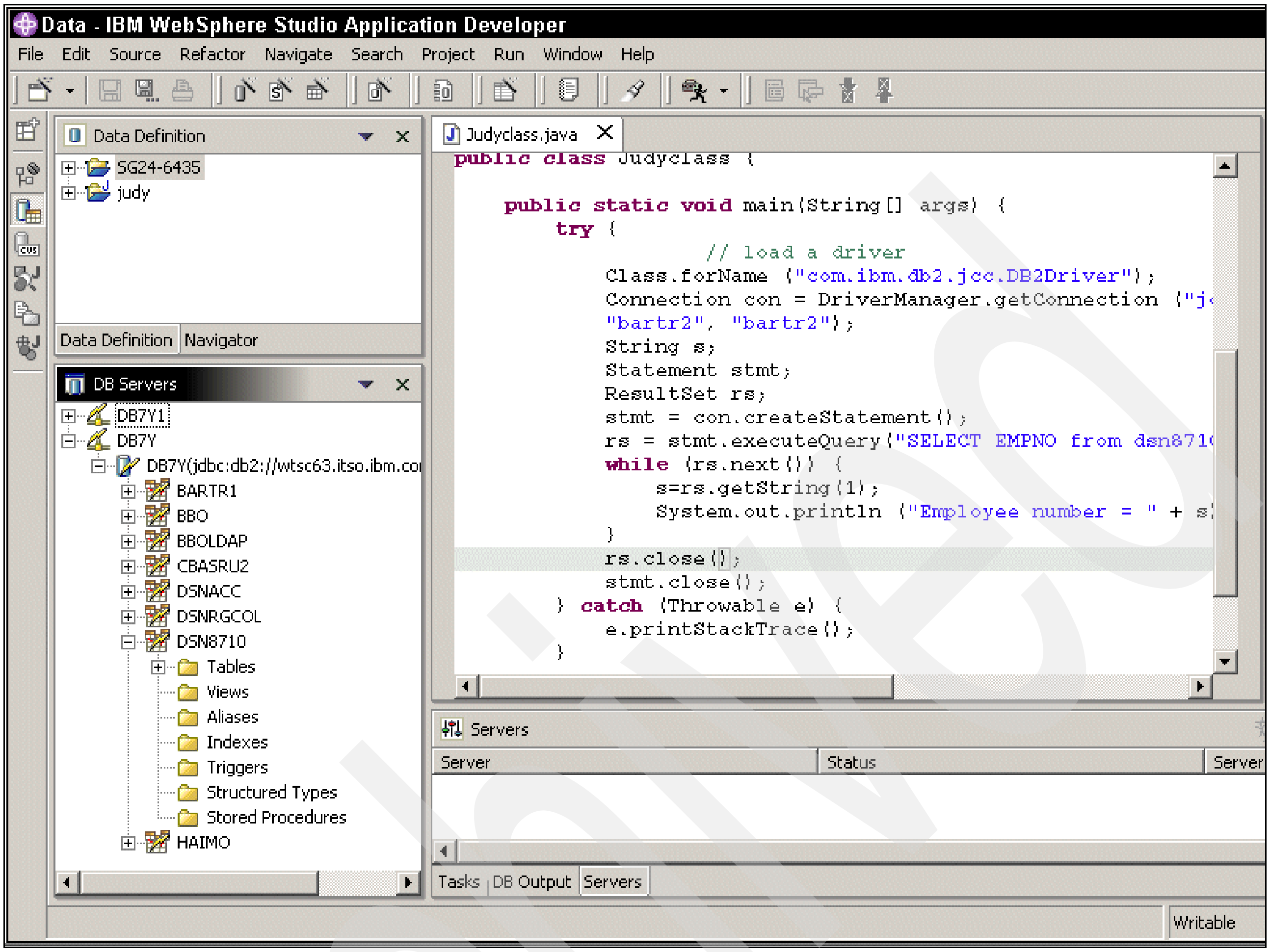

Figure 5-20 The databases from DB7Y

You see the actual URL that we just entered when creating the connection. Expand the URL by clicking its **+** sign, and the schemas are shown.

> **Important:** WSAD utilizes the database metadata catalog queries and message formatting methods when connected to DB2 UDB V7 for OS/390 and z/OS that we described in “Required DB2 for z/OS changes to enable the Universal Driver” on page 58. If you do *not retrieve any schemas* (there will be no error message) when you expand the URL, it is an indication that the PTF named in “Required DB2 for z/OS changes to enable the Universal Driver” has either not been applied, or that the customization job described in that section has not been run.

As we continue to expand the DSN8710 schema, we see the tables, views and other objects displayed. The DB2 for OS/390 and z/OS V7 sample database is created by running the DB2 installation job *hlq*.NEW.SDSNSAMP(DSNTEJ1).

> **Important:** If all you *see are the schemas with no error message but no expansion*, you probably do not have DSNZPARM DESCSTAT set to YES (see “DSNTIJUZ” on page 59).
>
> We found out the hard way. YES is not the default (in Version 7). Rerun the DSNTIJUZ job to update the value, then issue -SET SYSPARM LOAD(*DSNZPARM*) to load the changed DSNZPARMs.

To give you a feel of what information is available in the WSAD data perspective, we expand the Tables folder, and the DB2 tables are displayed as Figure 5-21 illustrates.

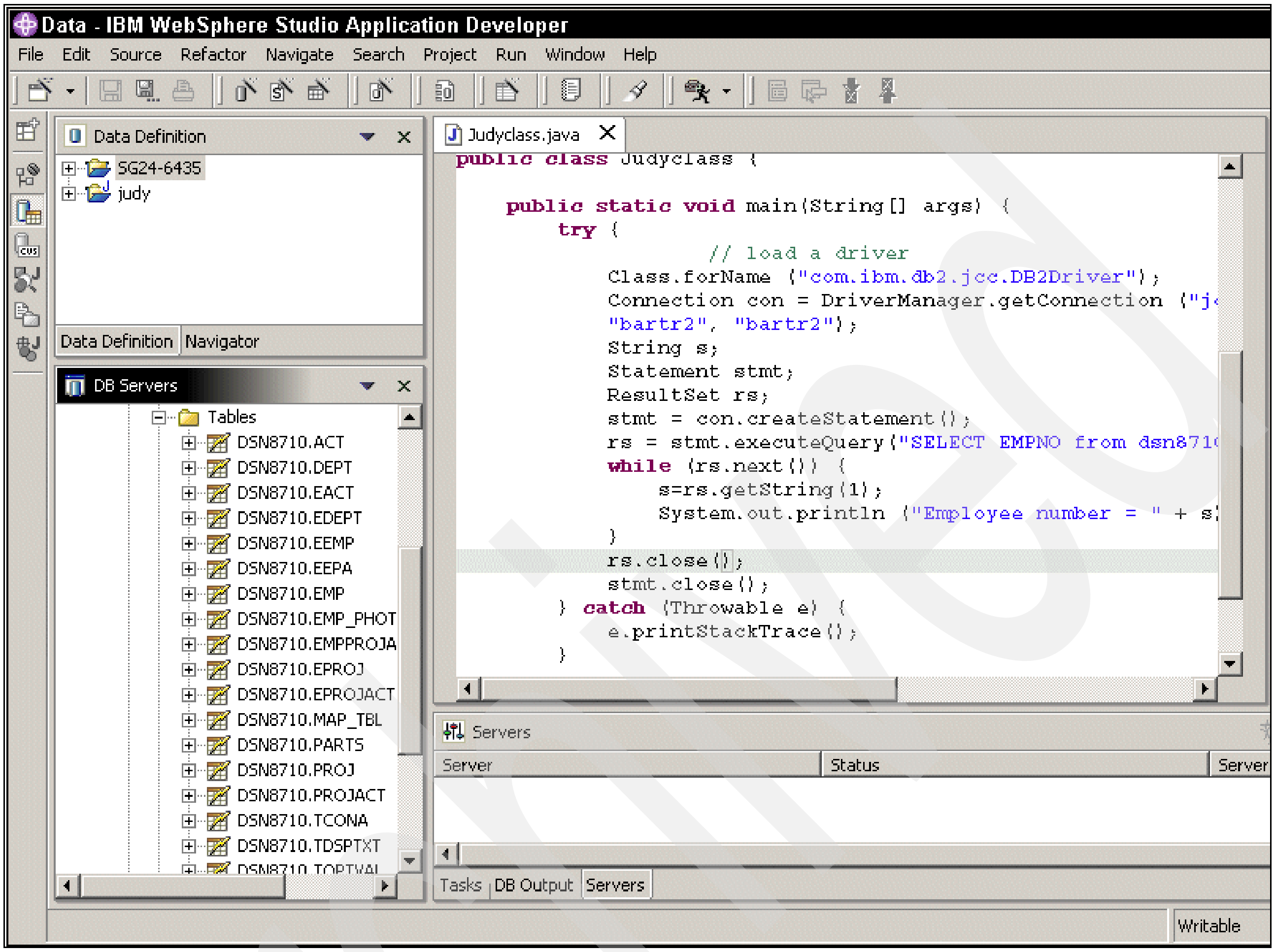

Figure 5-21 The tables in DSN8710 database

We click **DSN8710.EMP** and the window in Figure 5-22 on page 82 appears.

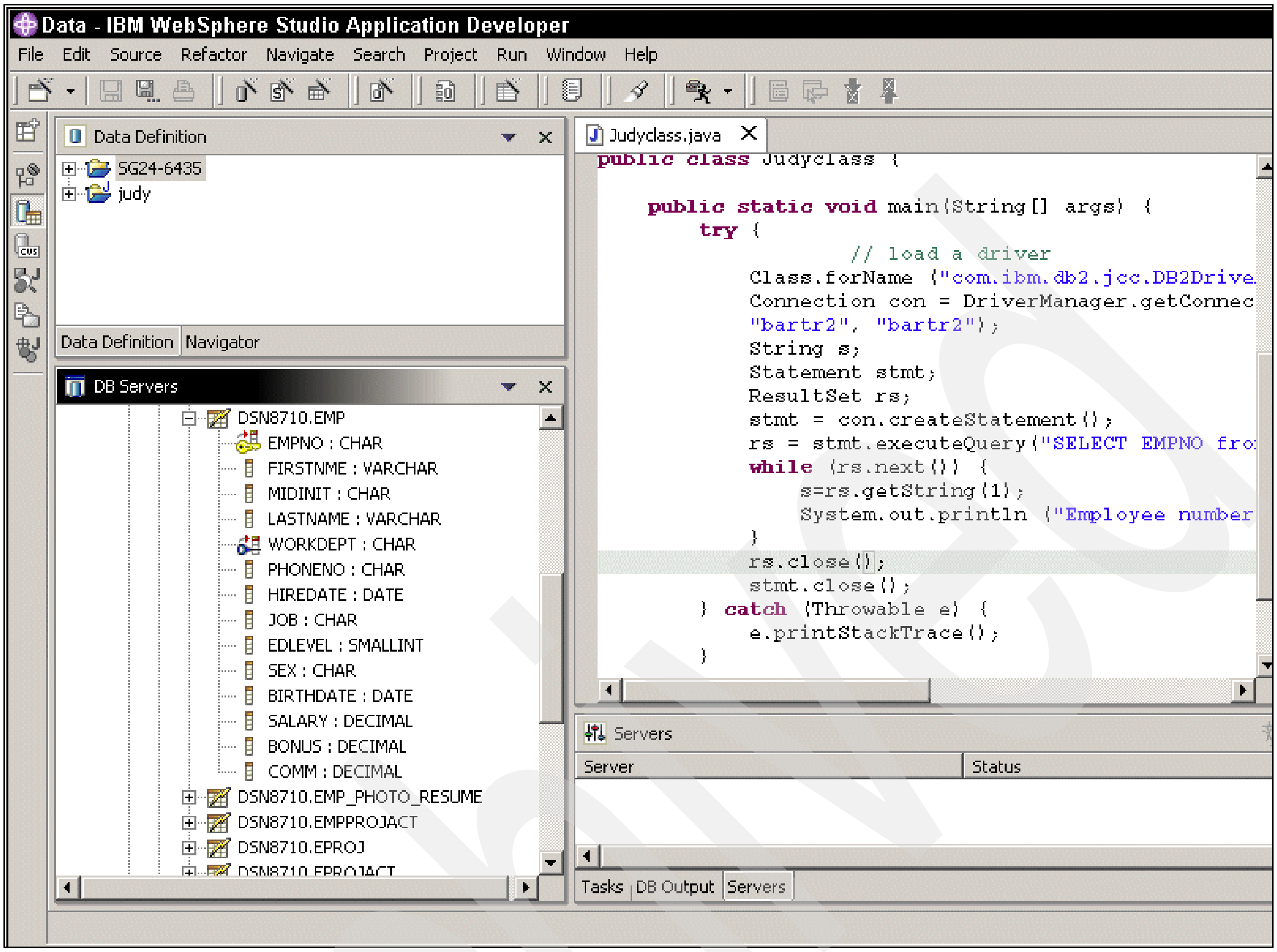

Figure 5-22 The columns of the sample EMP table

You can see the column names, with their data types, as well as primary and foreign key information. These columns may be used directly by WSAD application developers to ensure that the right column name and type is used in a program. You can also look at the sample contents of the table by right clicking the **DSN8710.EMP** table, and selecting **Sample Contents** from the drop-down box.

5.6 WebSphere for z/OS datasource setup

In this section we do not describe the entire setup of a WebSphere Application Server on z/OS. We assume that this has been taken care of by your systems programmer. In this section we focus on how to set up a datasource in WebSphere, so that it can be used by our (Servlet) application that we will develop in Chapter 15, “Using Servlets to access DB2” on page 249.

As mentioned before, we use the brand-new IBM DB2 Universal Driver for SQLJ and JDBC. Because our WebSpere runs on z/OS, we want to use a JDBC Type 2 driver to connect to the DB2 system (DB7Y) that runs on the same z/OS image.

To set up a data source in WAS, we use the WAS for z/OS V5 Administrative Console. (This browser appliation replaces the SMEUI that was used in previous versions of WAS on z/OS.)

5.6.1 Log onto the WAS Administrative Console

The WAS admin console uses a Web interface. Open a browser session and type in the URL to connect to the admin console application. In our case the URL is:

```
http://wtsc63.itso.ibm.com:9080/admin/
```

A screen similar to Figure 5-23 should appear.

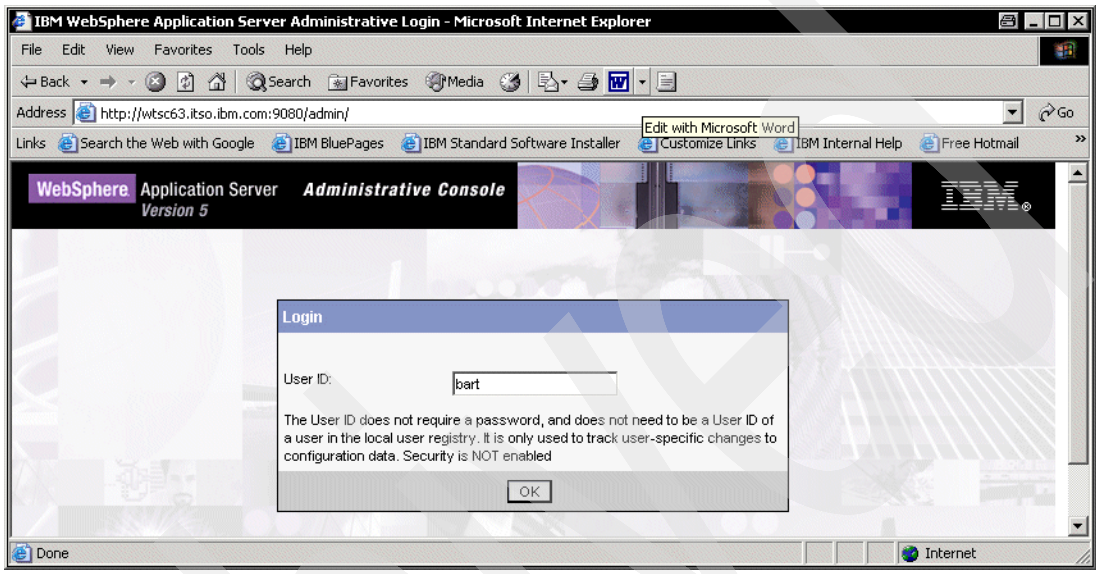

Figure 5-23 WAS Administrative Console

Type in your user ID and click **OK**.

5.6.2 Setting up system variables

This takes you into the Administrative Console application (Figure 5-24).

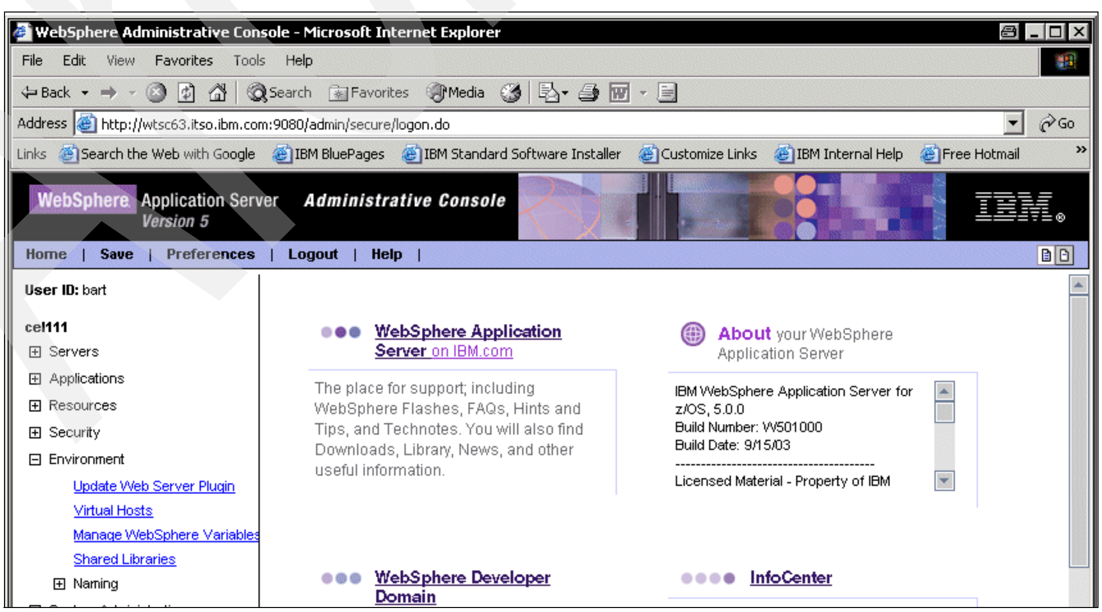

Figure 5-24 WebSpere Administrative Console

Before we can start with the definition of the DB2 datasource, we first need to set up a few system variables inside WAS. To do so, expand the **Environment** tree in the left pane, and select **Manage WebSphere Variables**. This takes you to the window shown in Figure 5-25.

Make sure that the scope is Node (indicated by the red arrow). If not, select **Node** and click **Apply**.

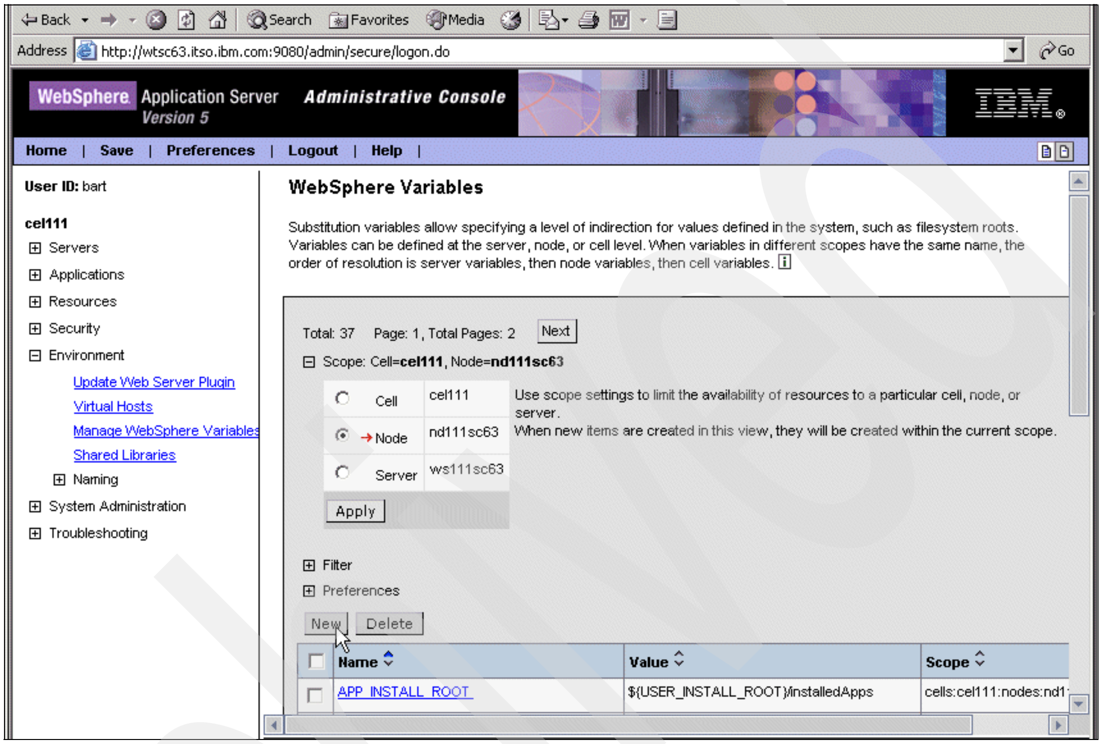

Figure 5-25 Create a new variable

DB2JCC_DRIVER_PATH variable

Then click **New** to define a new variable. This takes us to a screen like Figure 5-26 on page 85 where we specify the name of the variable DB2JCC_DRIVER_PATH, a description, and its value /pp/p/db2javadriver/. This is the path where the JCC driver is installed into the USS file system (HFS). Click **Apply** when done.

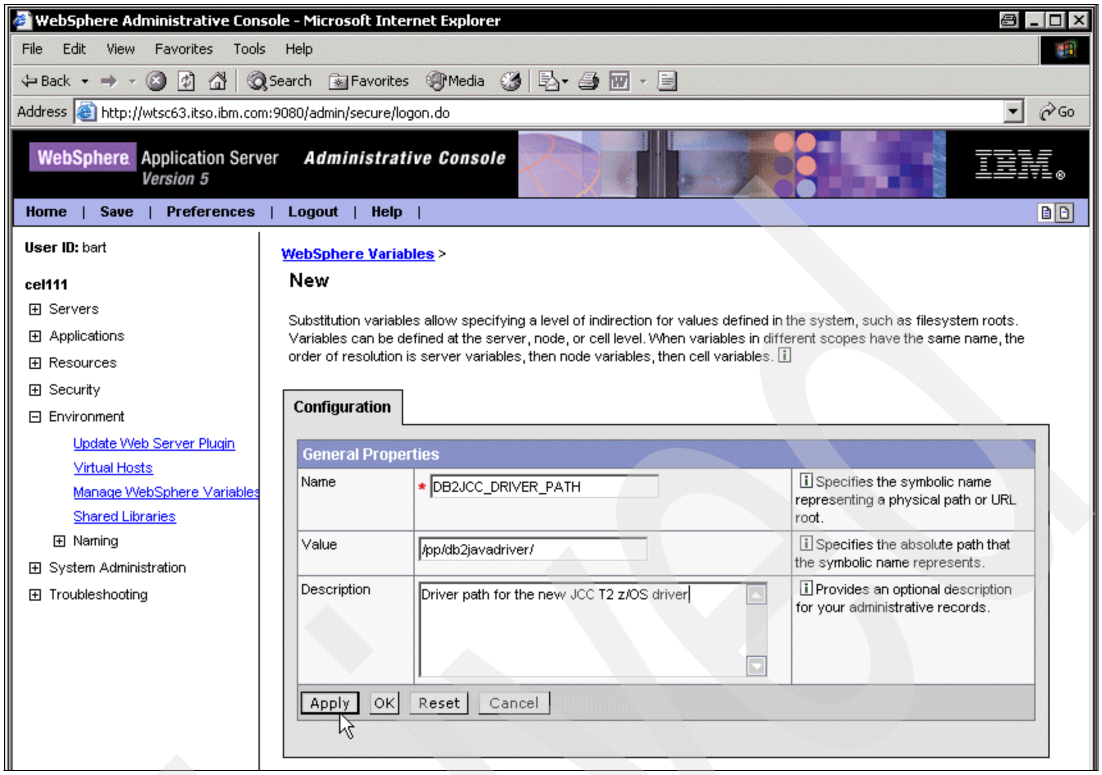

Figure 5-26 Create new WebSphere variable

Because we made a change to the configuration, we have to save the change. Click **Save**, as shown in Figure 5-27 on page 86.

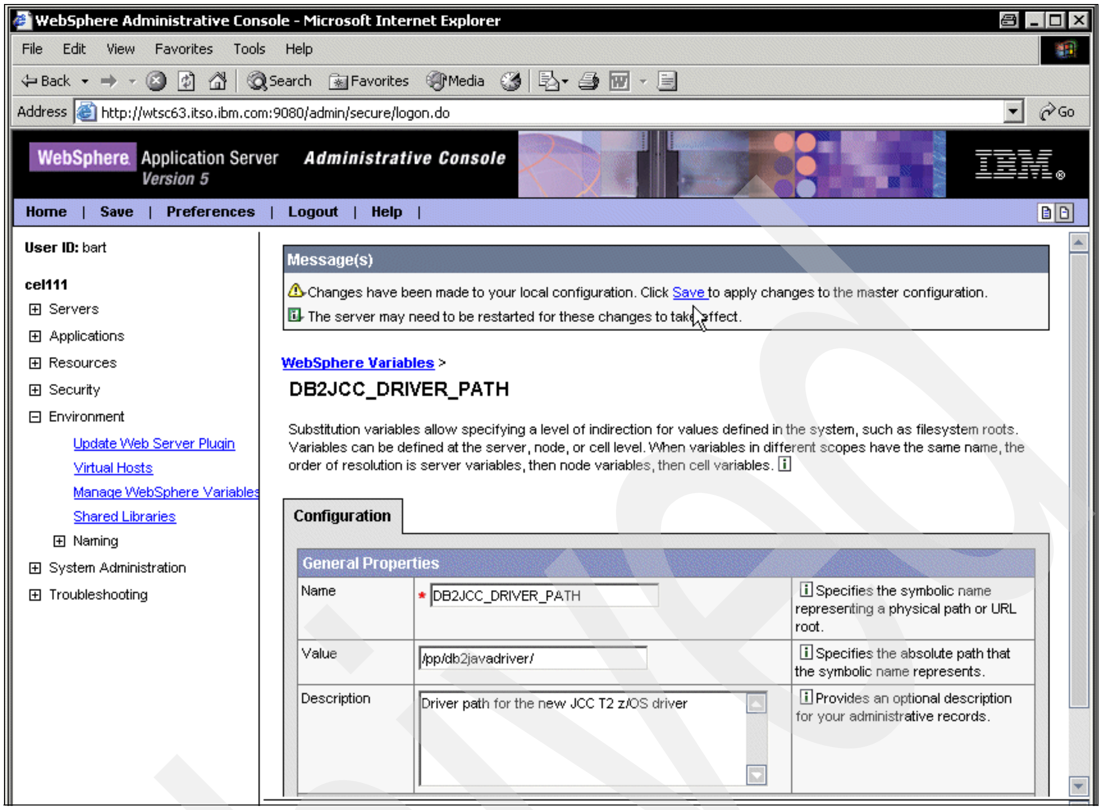

Figure 5-27 Save your changes

Save into the master configuration as well. Click **Save** again, as indicated in Figure 5-28.

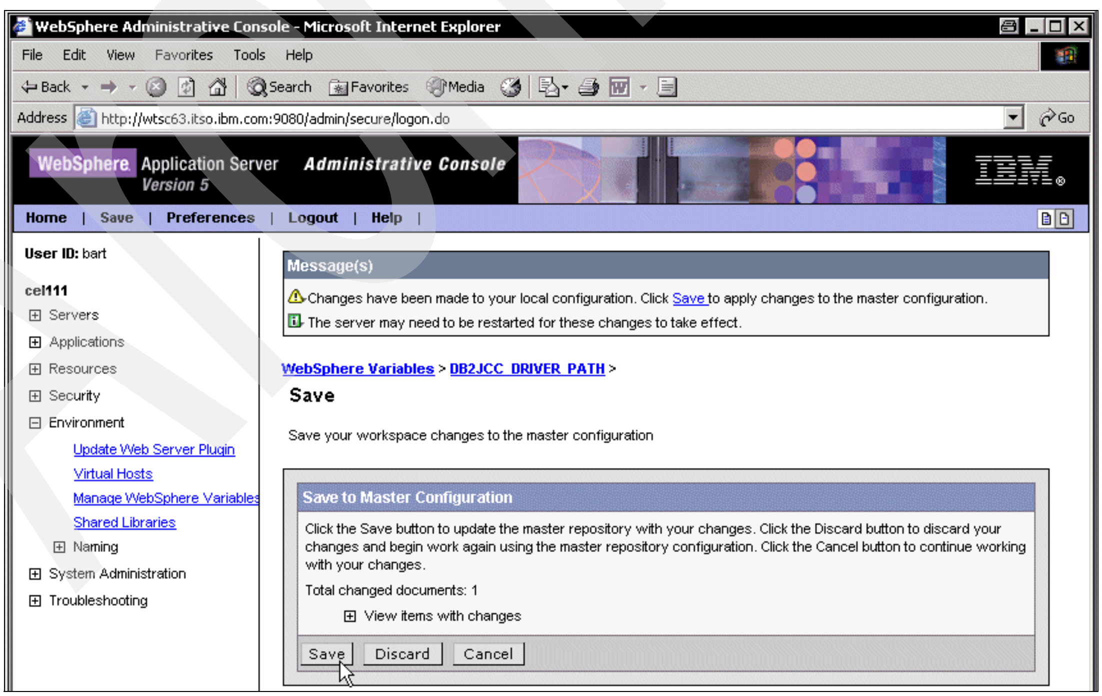

Figure 5-28 Save into the Master Configuration

DB2JCCPROPERTIES variable

Now do the same thing again, and define an additional variable to indicate where the JCC driver properties are stored.

Variable DB2JCCPROPERTIES.

Value ${DB2JCC_DRIVER_PATH}/db2jcct2.properties. You must specify the full path name for the properties field. Note that we use the variable that we previously defined. The file name itself must be db2jcct2.properties.

> **Attention:** As mentioned before, we use a beta version of the JCC driver on z/OS. It is very likely that there will be changes in the way you specify properties for the JCC driver in the GA version of the code. Please refer to the appropriate GA documentation for details.

Setting up a JDBC provider

Now we are ready to set up our JDBC provider definition. Before we begin, it is important to note that if you use both the old DB2 390 driver and the new DB2 JCC driver under WAS for z/OS, you need to ensure that DB2 JDBC providers associated with these two drivers are not inter-mixed in the same server. A given server may only have DB2 JDBC providers of a single type: Either the DB2 390 JDBC driver type or the DB2 JCC JDBC driver type. Because of this, you should restrict all DB2 JDBC provider defintions for these two drivers to a scope of "server" so you can better manage the DB2 providers to ensure they are not inter-mixed in a server. The reason for this restriction is that there are identical class names in both drivers.

To use the new DB2 JCC JDBC driver, you need to create a new JDBC provider that is associated with the new driver. To do so using the WAS Administrative Concole, go to **Resources** (left pane) **-> JDBC Providers** (Figure 5-29).

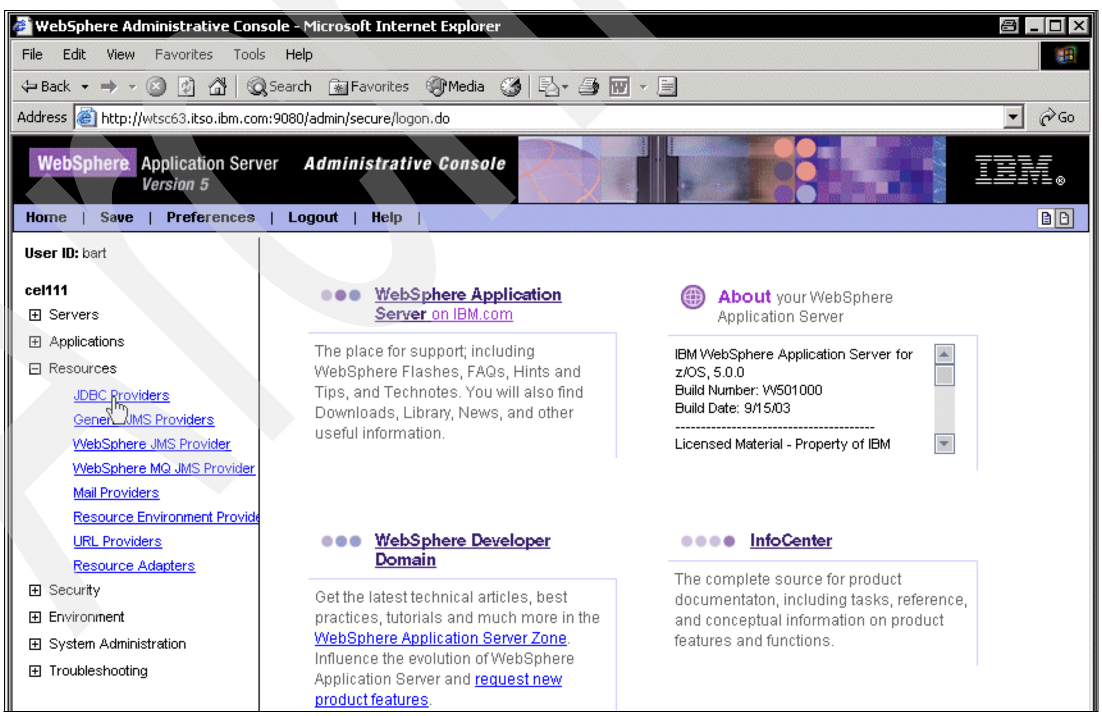

Figure 5-29 JDBC providers

A screen frame similar to Figure 5-30 will be displayed. To specify the scope of the provider, select **Server** and then **Apply**. Once the scope is defined to the server, select **New** to create a new JDBC provider.

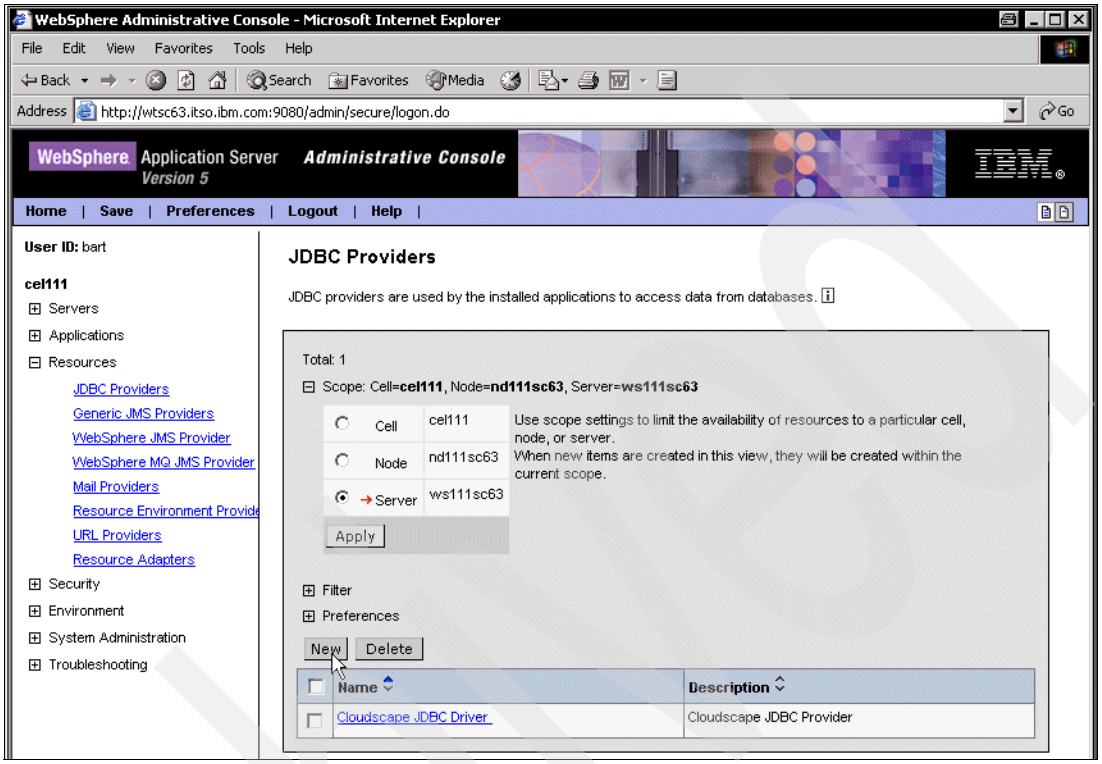

Figure 5-30 Create a new JDBC provider

A window similar to Figure 5-31 on page 89 is shown. Select **User-defined JDBC provider** from the drop-down list and click **Apply**. We have to use this option because we use a brand-new driver. Once the JCC driver is GA, WAS will predefine some settings for the driver, and you will be able to select it from the drop-down list. (Note that DB2 390 Local JDBC Provider (RRS) should not be selected. This is not the JCC driver but the old (current T2 driver that ships with V6/V7).)

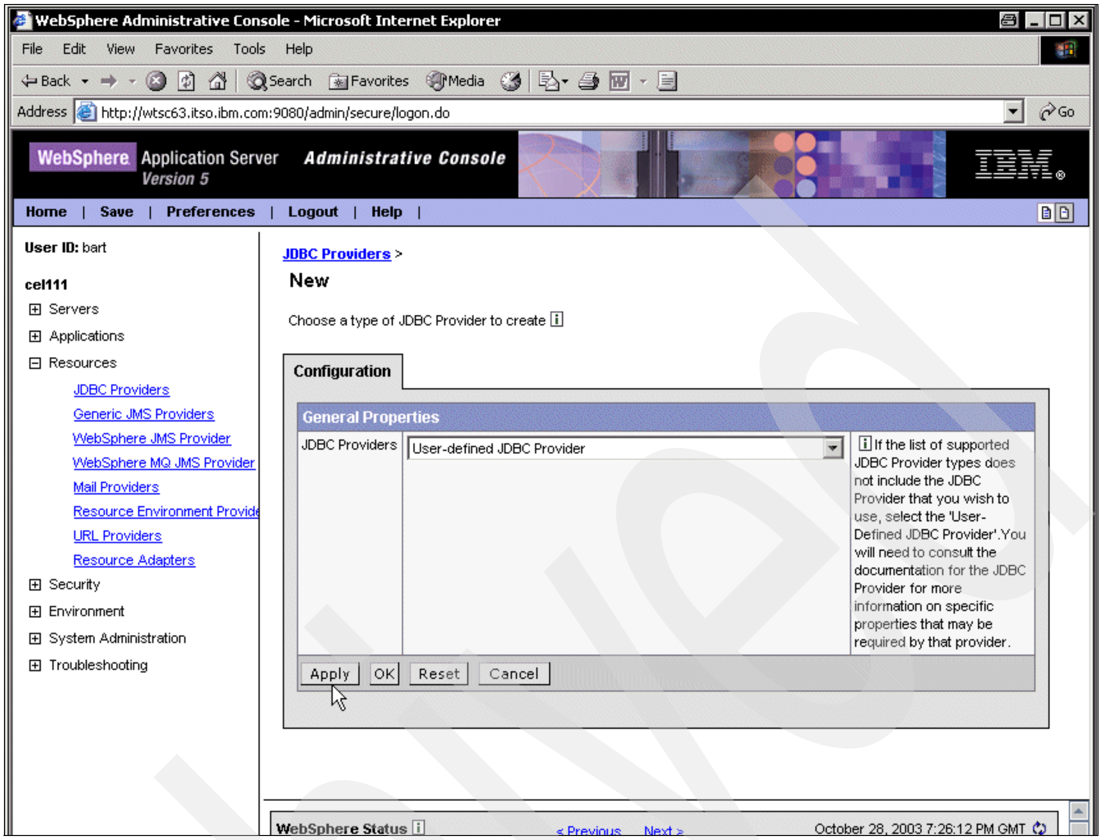

Figure 5-31 Select user-defined JDBC Provider

Click **Apply** to create a user-defined JDBC Provider. A JDBC Provider screen frame with empty fields is displayed. Fill in the fields in as indicated below.

Name	DB2 JCC T2 zOS JDBC Provider
Description	Custom JCC T2 configuration
Classpath	${DB2JCC_DRIVER_PATH}/classes/db2jcc.jar ${DB2JCC_DRIVER_PATH}/classes/db2jcc_license_cisuz.jar ${DB2JCC_DRIVER_PATH}/classes/sqlj.zip
Native Library Path	${DB2JCC_DRIVER_PATH}/lib
Implementation Classname	com.ibm.db2.jcc.DB2ConnectionPoolDataSource

This is also shown in Figure 5-32 on page 90 and Figure 5-33 on page 90.

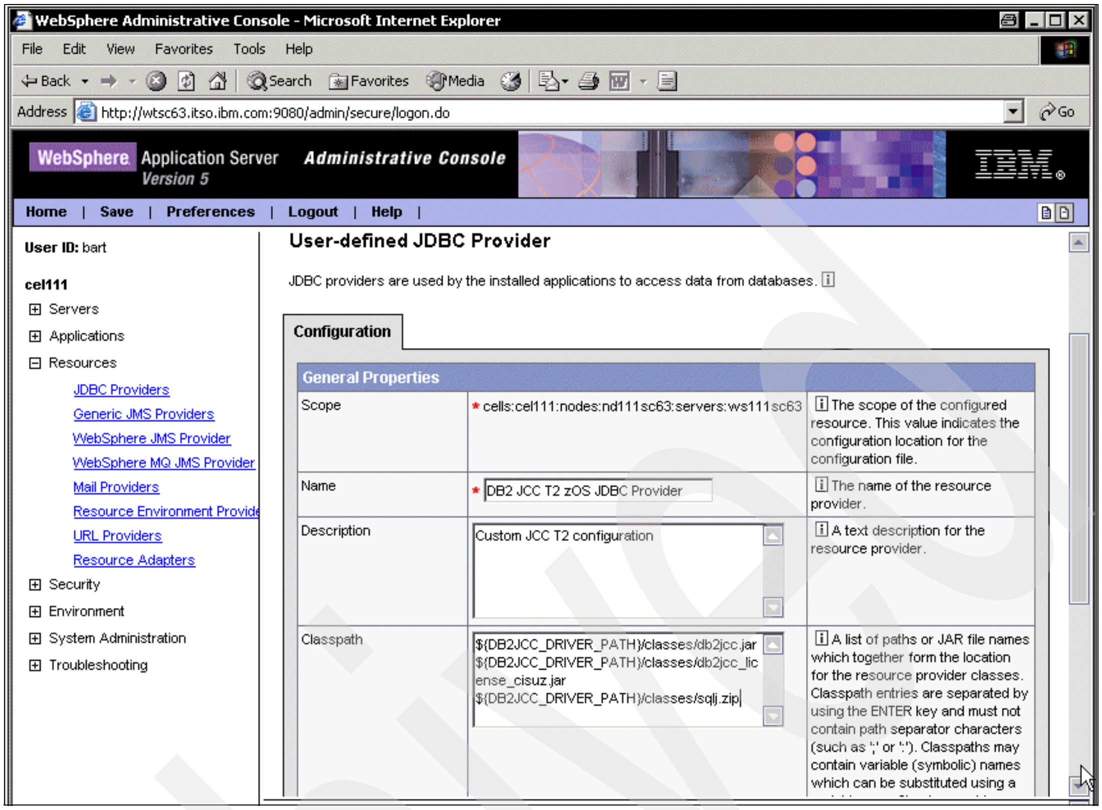

Figure 5-32 User-defined JDBC Provider variables

Once you have filled in this information, click **Apply** and then **Save** to complete the creation of the new JDBC provider.

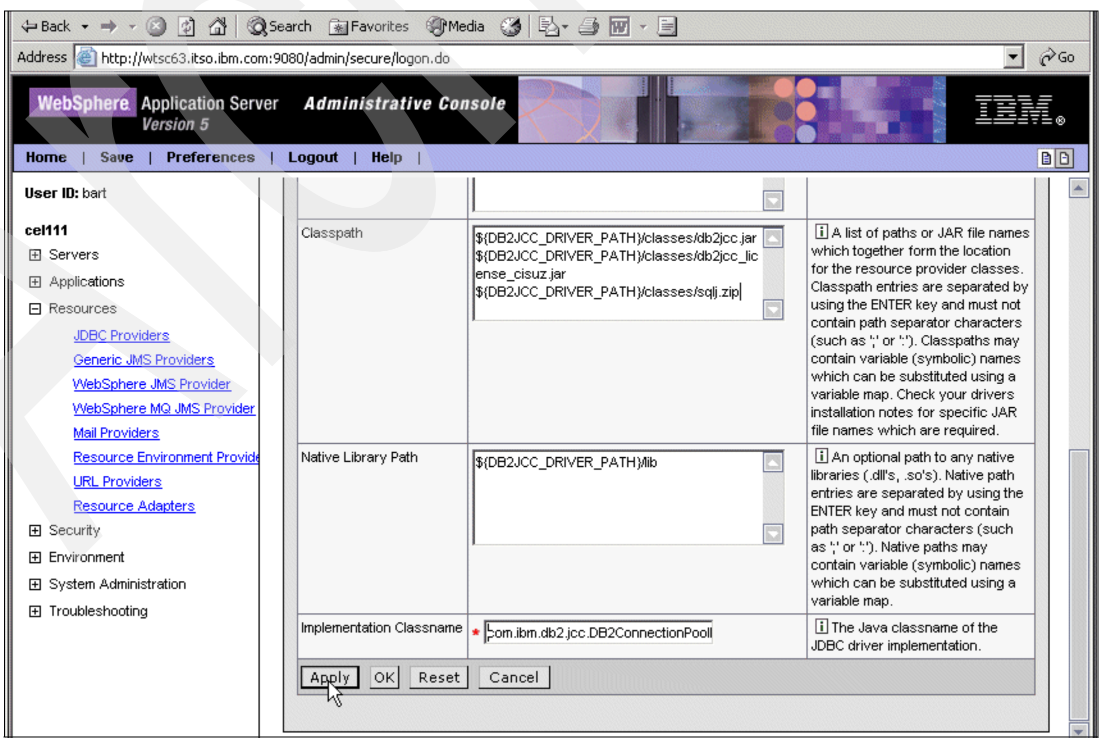

Figure 5-33 User-defined JDBC Provider variables - 2

Defining the data source

We are now ready to create a data source for the new DB2 JCC T2 z/OS JDBC Provider we created in the previous section. Using **Resources -> JDBC Providers**, select **DB2 Jcc T2 zOS JDBC Provider**, which is now listed as an available provider. Click this provider to display it (Figure 5-34).

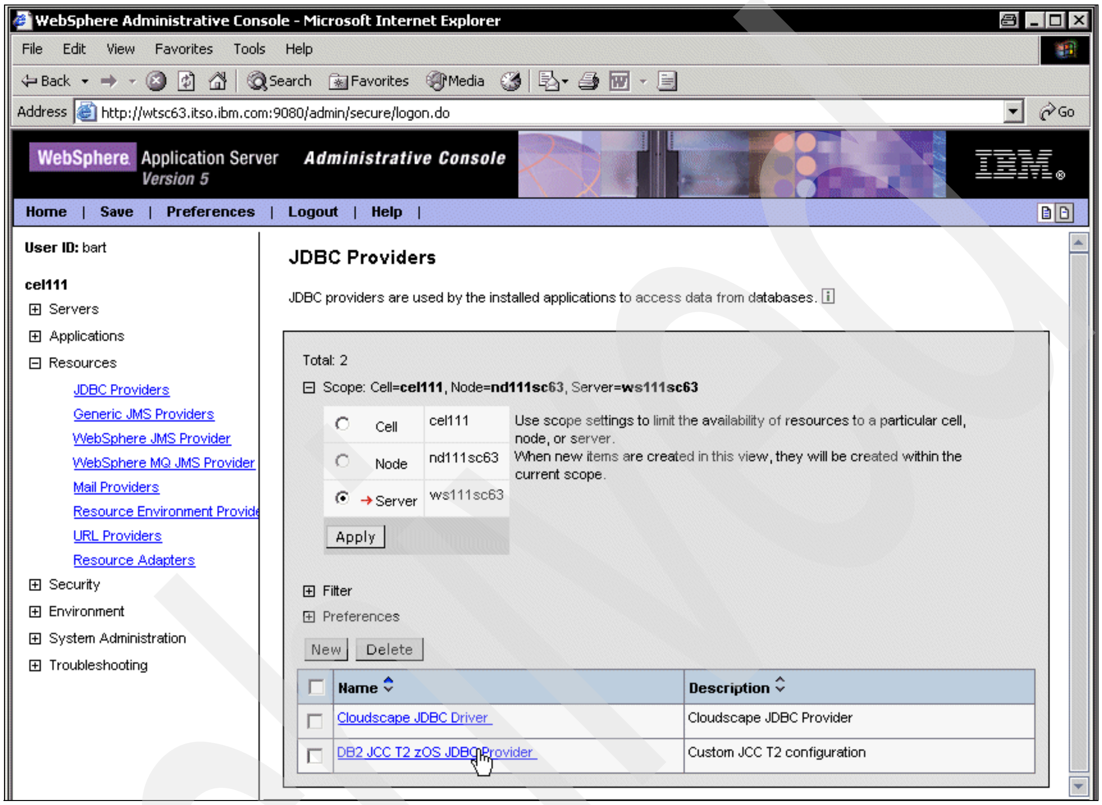

Figure 5-34 Display the JDBC provider

Scroll to the bottom of the resulting window, and click **Data Sources** (Figure 5-35 on page 92).

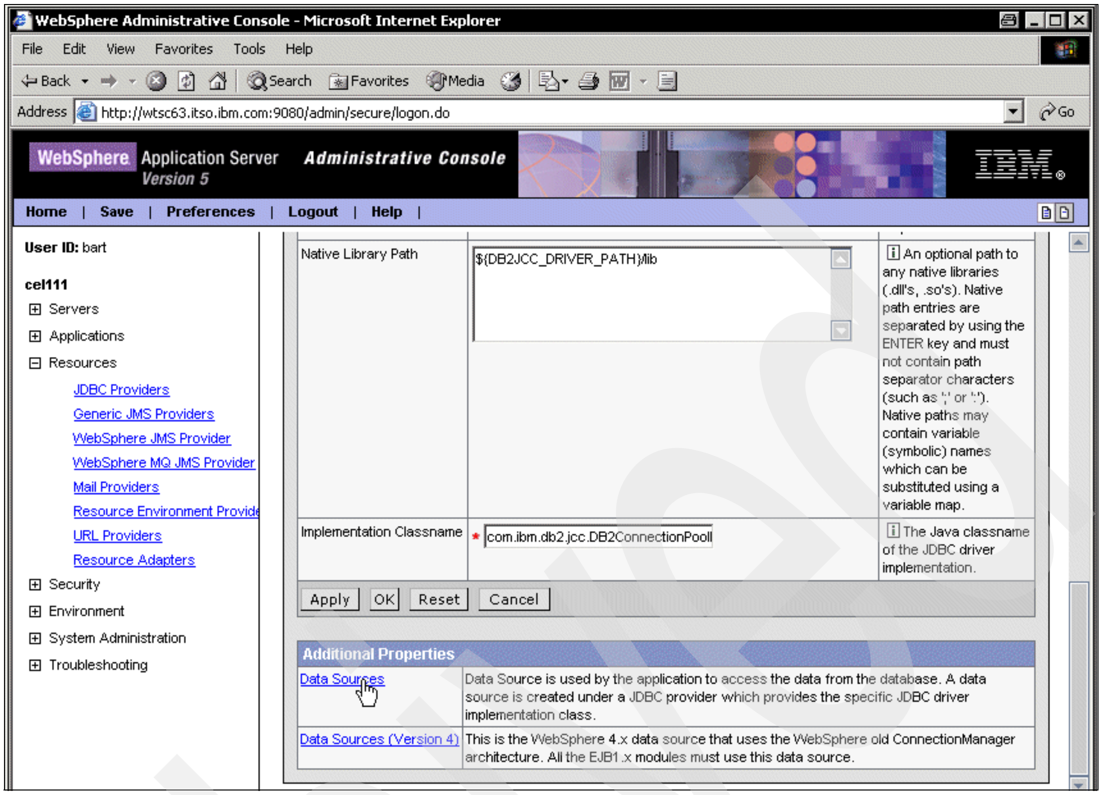

Figure 5-35 Select data sources

A JDBC Providers -> DB2 JccT2 zOS JDBC Provider -> Data Sources screen with no data sources is displayed. On this screen, select **New** to create a new datasource (Figure 5-36).

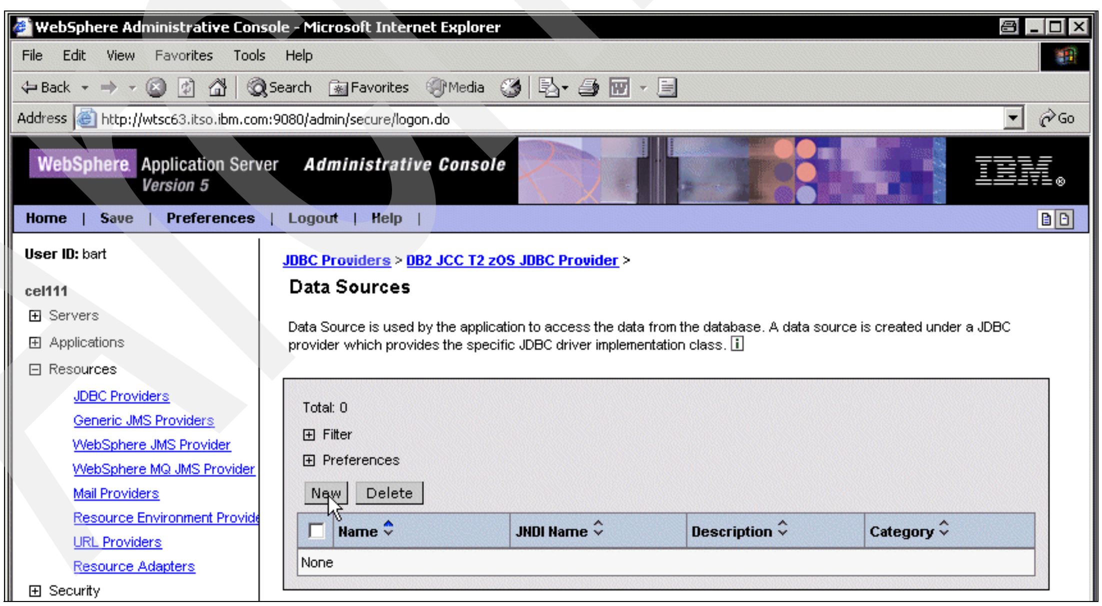

Figure 5-36 Create new data source

This takes you to the JDBC Providers -> DB2 JccT2 zOS JDBC Provider -> Data Sources New screen. Initially all the fields except the "Statement Cache Size" and the "Datasource Helper Classname" fields are empty. The Statement Cache Size field is defaulted to 10 and the "Datasource Helper Classname" is defaulted to a class name com.ibm.websphere.rsadapter.DB2390LocalDataStoreHelper. The "Datasource Helper Classname" field should not be changed. Fill in the following fields as indicated below:

Name	DB2JccT2zOSDataSource
JNDI Name	jdbc/empdb
Description	New JDBC Datasource
Container managed persistence	<check the box on the screen>

You may also fill in the Container-managed Authentication Alias field if you want to specifiy an authentication alias that your installation has set up to contain the user ID and password that you want associated with all connections obtained from the datasource. If you do not specify anything in this field and if the resource reference of this datasource gets bound to in an application that is defined with resauth=Container, all connections obtained from the datasource will be associated with the server's identity.

The following screen shots (Figure 5-37 and Figure 5-38 on page 94) are a partial view of what the new datasource screen looks like when filled out.

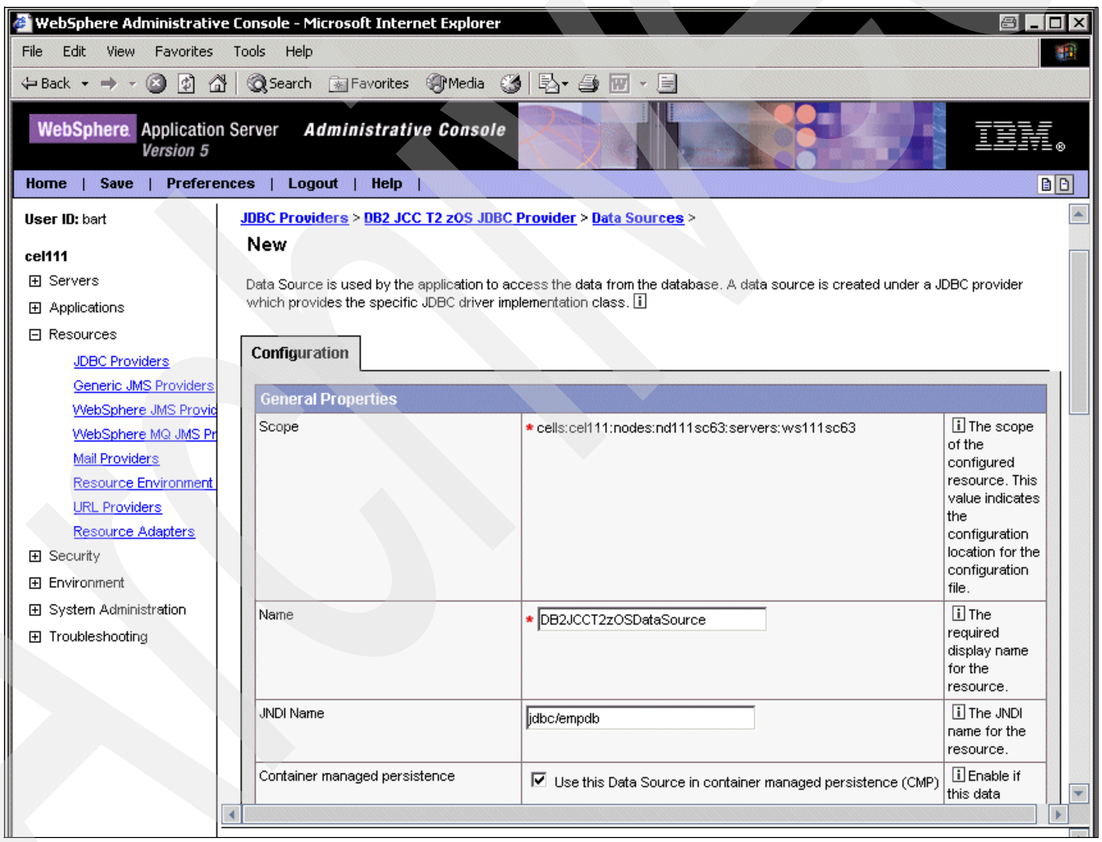

Figure 5-37 Setting up a datasource definition

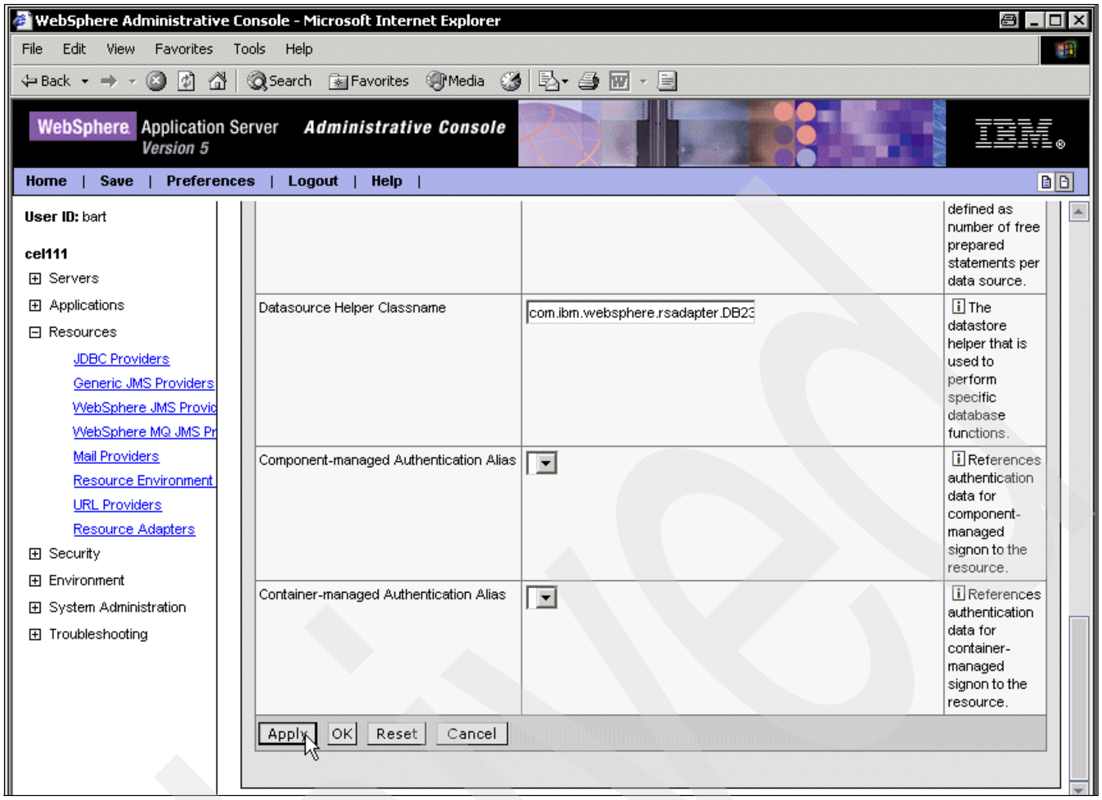

Figure 5-38 Setting up a datasource definition - 2

Click **Apply**, but do not immediately save the new definition. Before saving, go to the bottom of the new datasource screen and click **Custom Properties** to define the datasource properties (Figure 5-39 on page 95).

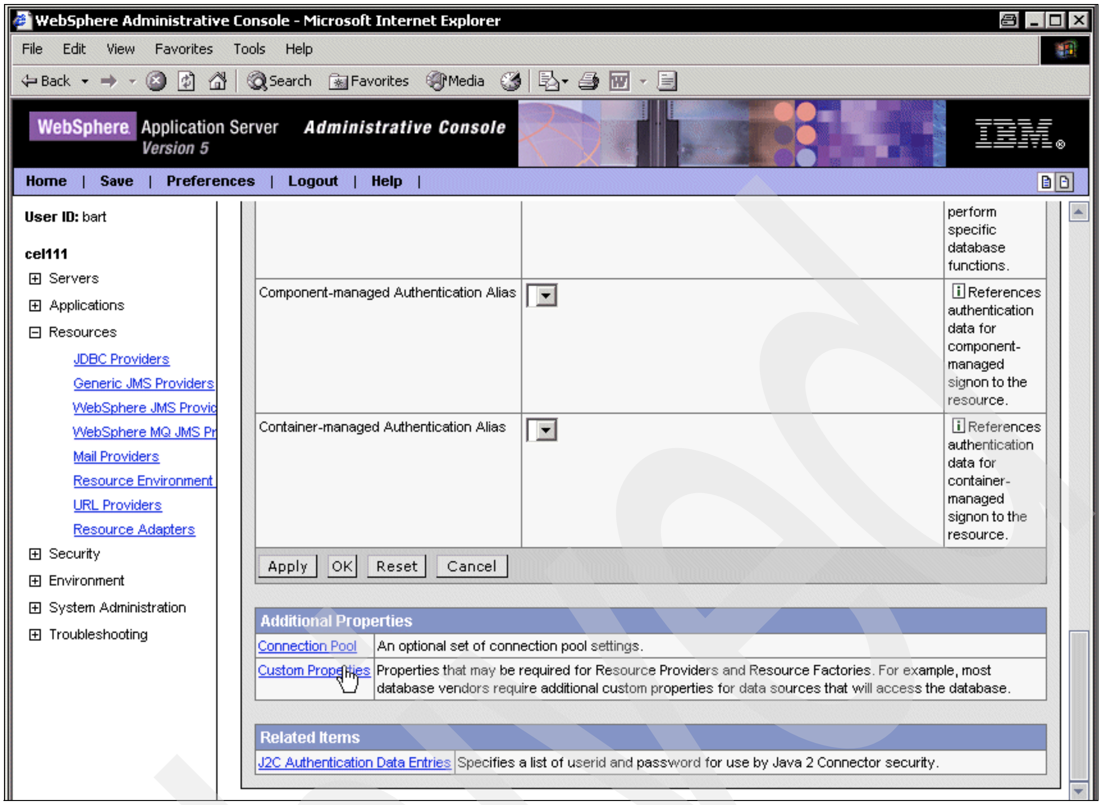

Figure 5-39 Select custom properties

When you do this, the Custom Properties window is displayed (Figure 5-40 on page 96). You need to provide some values for some of the variables.

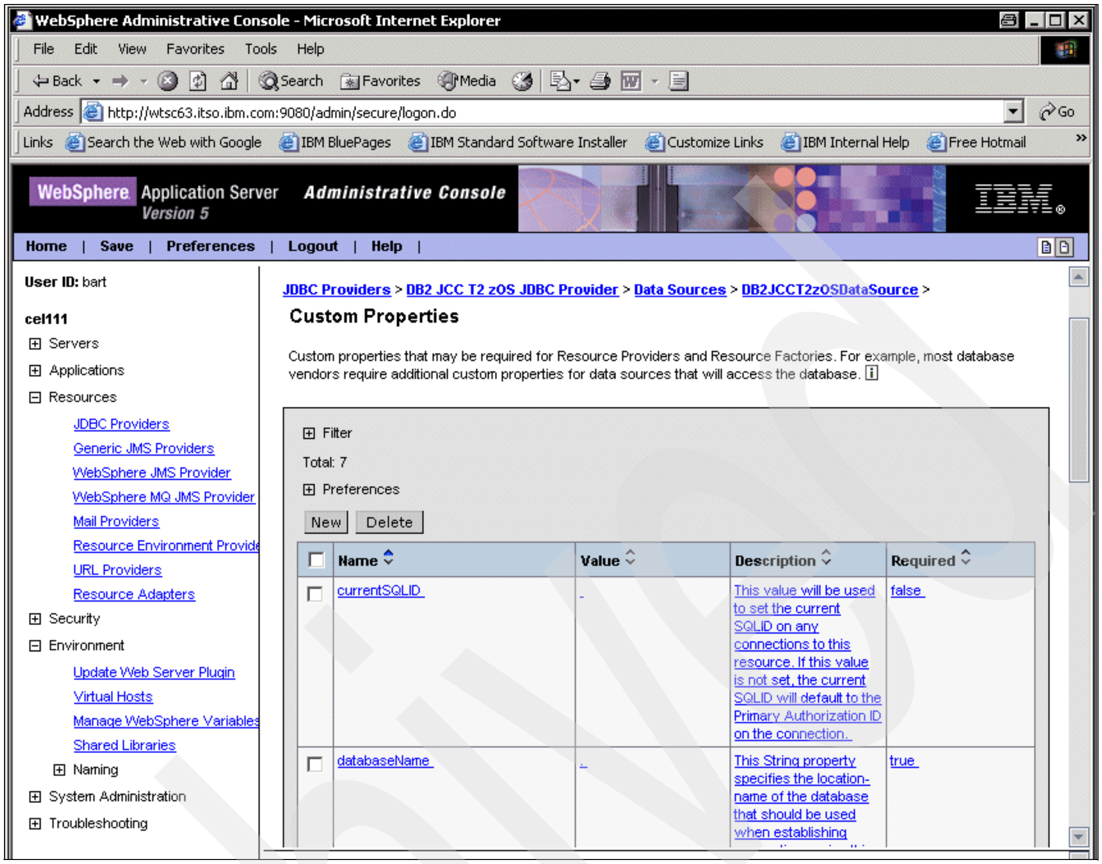

Figure 5-40 Custom Properties window

Fill in the values for each of the fields as indicated below.

databaseName DB7Y, which is the DB2 location name.

Description New JDBC Datasource.

loginTimeout 0.

planName This field is initially set to DSNJDBC. Change it to BARTSQLJ, our SQLJ plan.

You do so by clicking the name of the variable. We show how to specify the databaseName in Figure 5-41 on page 97.

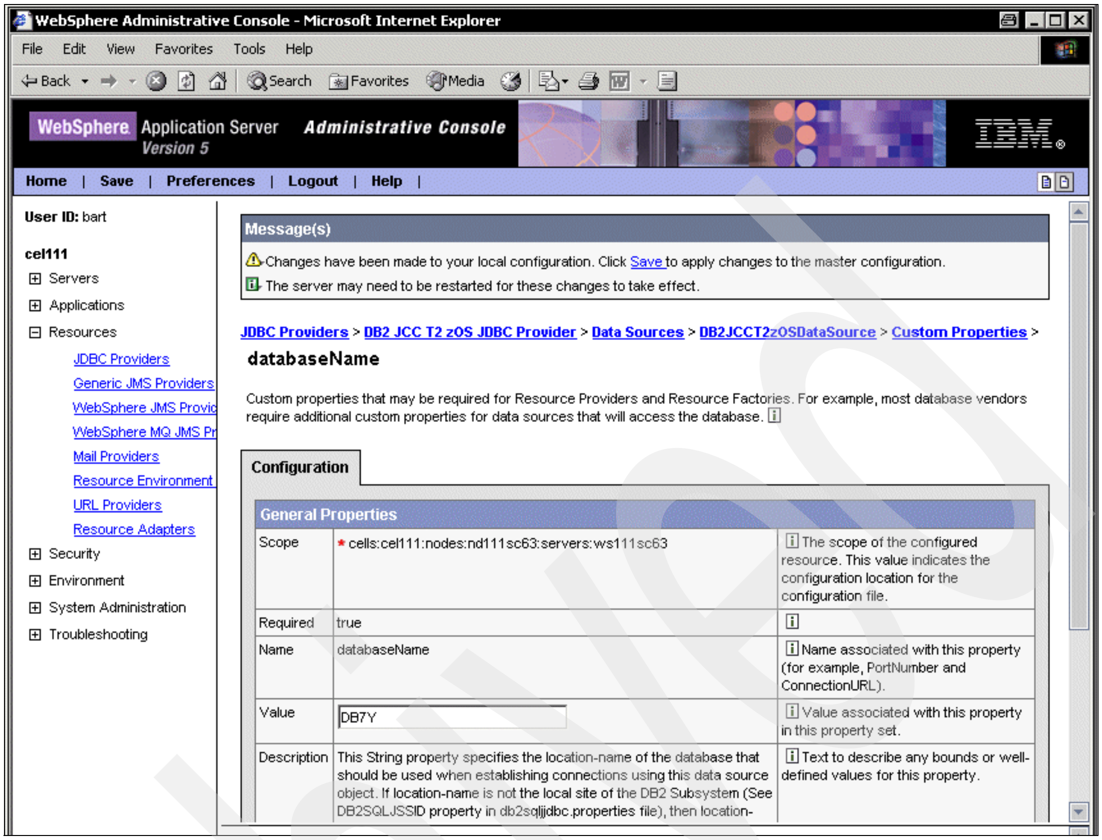

Figure 5-41 Specifying the databaseName

When done, click **Ok** and continue with the other variables. When you are done, you can **Save** the new datasource definition.

In addition to the standard DB2 JDBC datasource properties that are initially defined for the datasource, you can also add other properties to reflect new properties that are now supported by the new DB2 JCC datasource implementation. To do so, using the administrative console, go back to the JDBC provider datasource you just created and click **Custom Properties**. The Custom Properties screen is displayed again. For each new property you wish to add, click New. Once you click New, a screen is displayed where you can enter the new Property Name, Property Description, Property Value, and Property Type. After entering all the property information, click **Apply**. Repeat this process for each property you wish to add. After applying the last property you wish to add, click **Save** to save all the new datasource custom properties. We did not add any additional properties for our simple application.

To make sure the datasource definition gets activated, we bounced the WAS Server.

Part 3

Putting it all together

In the previous parts of this publication we described the concepts and facilities for accessing DB2 from Java, and how to set up your environment.

In the following chapters we take you through a step-by-step description, much like a cookbook, of how to actually use all the different parts and run your first Java application to access DB2 data. We do this for both JDBC and SQLJ applications.

Also, we provide a reference on SQLJ syntax and usage, and SQLJ tools.

We explain how to access DB2 from Web applications, and address performance topics.

6

Getting started with JDBC

Traditionally, when you learn a new programming language or environment, you start with a "Hello World" kind of program to get familiar (and, we hope, a feeling of success). In this chapter, we do just that, writing a very simple JDBC program and running it from both the workstation and the z/OS USS environment.

6.1 Creating the project

In this section we write and prepare the code for our simple test program. The program reads the DSN8710.EMP DB2 sample table, retrieving all employee records where the salary is in a given range and prints each row retrieved.

To develop our sample program we use WebSphere Studio Application Developer, the IBM Java development platform. In WSAD, any code you write must reside in a *project*. A project is the top level of organization of your resources in the WSAD workbench. A project contains files and folders. Projects are used for building, version management, sharing, and organizing resources.

1. Start WSAD and create a new project called SG24-6435: Select **File -> New -> Project…**, then select **Java** in the left pane and **Java Project** in the right pane. Press **Next**. On the next panel, enter the project name (`SG24-6435`). You should leave the checkbox “Use default project contents” checked.
   Press **Finish**. WSAD automatically changes to the Java perspective.
2. Create a Java package for our program to live in. In the Package Explorer view, select the project name, then select **File -> New -> Package** (or press the New Java package icon () on the toolbar). Enter the package name, `com.ibm.itso.sg246435.jdbc`, and press **Finish**.

   WSAD creates subdirectories under your project contents directory to reflect the package name. That is, we now have a directory called:

   WSDir`\SG24-6435\com\ibm\itso\sg246435\jdbc`

   Where *WSDir* is your WSAD workspace directory.
3. Now we create the Java source file for our Hello program. In the Package Explorer view, select the package then select **New -> Class**. The New Class wizard opens (see Figure 6-1). In the Name field, type `Hello`, check the appropriate “Which method stubs would you like to create” boxes, and press **Finish**.

Figure 6-1 New class wizard

4. In the editor window that opens, type the source code for our Hello application as shown in Example 6-1, changing the values for *url*, *user* and *password* accordingly.

Example 6-1 Source code for Hello application

```
package com.ibm.itso.sg246435.jdbc;
import java.math.BigDecimal;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * Sample test program to retrieve all employees
 * in the EMP sample tables whose salary is in
 * a given range.
 *
 * @author Ulrich Gehlert
 */
public class Hello {

  /** JDBC URL to the database. */
  private static final String url = "jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y"
                                  + ":retrieveMessagesFromServerOnGetMessage=TRUE;";     1

  /** User name to connect to the database. */
  private static final String user = "bartr1";

  /** Password for the database connection. */
  private static final String password = "b1ngo";

  public static void main(String[] args) {
   Connection conn = null;
   PreparedStatement stmt = null;
   ResultSet rs = null;
   try {
     // Load the JDBC driver.
     Class.forName("com.ibm.db2.jcc.DB2Driver");                                      2

     // Connect to the database server.
     conn = DriverManager.getConnection(url, user, password);                         3

     // Prepare the SELECT statement.
     stmt = conn.prepareStatement(                                                    4
              "SELECT LASTNAME, FIRSTNME, SALARY"
            + "  FROM DSN8710.EMP"
            + " WHERE SALARY BETWEEN ? AND ?");                                       5

     // Set parameters for the SELECT statement.
     stmt.setBigDecimal(1, new BigDecimal(30000));                                    6
     stmt.setBigDecimal(2, new BigDecimal(50000));

     // Execute the query to retrieve a ResultSet.
     rs = stmt.executeQuery();                                                        7

     // Iterate over the ResultSet.
     while (rs.next()) {                                                              8
      String lastname = rs.getString(1);        // LASTNAME
      String firstname = rs.getString(2);       // FIRSTNME
      BigDecimal salary = rs.getBigDecimal(3);  // SALARY
```

```
        System.out.println(lastname + ", " + firstname + ": $" + salary);
      }

    } catch (SQLException e) {
      // Print exceptions to the console.
      System.err.println(e.getMessage());
    } catch (Exception e) {
      System.err.println(e);
    } finally {
      // Clean up.
      try {
        if (rs != null) rs.close();
        if (stmt != null) stmt.close();
        if (conn != null) conn.close();
        } catch (SQLException ignored) {
        }
    }
  }
}
```

Notes® on Example 6-1 on page 103 (we cover these items in more detail in the following sections):

1. The `retrieveMessagesFromServerOnGetMessage` property, when enabled, directs all calls to the standard JDBC SQLException.getMessage() to invoke the server-side stored procedure (SQLCAMESSAGE), which retrieves the readable message text for the error. This property is disabled by default. You can also use the proprietary method DB2Sqlca.getMessage() to retrieve the fully formatted message text.
2. This line causes the JVM to load the class named in the argument, namely, the JDBC driver. As you can see, we load the new Universal Driver.
3. Establishes the connection to the database.
4. To prepare an SQL statement for execution, you invoke the `prepareStatement()` method of the Connection object.
5. The variable parts of the statement—in this case, the lower and upper limits for the SALARY column—are indicated by *parameter markers*, which, syntactically, are represented by question marks.
6. Before the statement can be executed successfully, we have to supply values for all parameter markers in the statement. Note that, in this example, we could have hardcoded the lower and upper bounds directly in the statement.
7. Populate the variable parts of the statement; in other words, supply values for the parameter markers.
8. This is where the statement is actually executed. To execute a statement that returns a result (such as the SELECT statement in our example), you use the `executeQuery()` method, whereas statements that do not return a result are executed via the `executeUpdate()` method.
   The `executeQuery()` method returns a `ResultSet` object, which allows you to iterate over the query results.
9. This is where we iterate over the result set. The `next()` method tries to fetch the next result row from the database, returning true if there actually was another result row, or false if no more rows were found. Thus, the loop terminates after the last row in the result set has been processed.

   We then print out the values in each column, referring to the columns by index.

6.1.1 Loading the JDBC driver

The first step before we can establish a connection is to load the JDBC driver. To dynamically load a Java class into the JVM, you use the `Class.forName()` method. This method does not have anything to do with JDBC per se; it can be used to load any class provided this class can be found on the application's runtime classpath. After the class has been loaded, the JVM initializes it; a class can provide its own initialization code by implementing a piece of code called a static initializer. In the case of a JDBC driver, the static initializer registers that newly loaded driver with the JDBC runtime.

6.1.2 Establishing the connection

Next, we can establish the connection to the database, supplying a JDBC URL, a user name and a password to the `DriverManager.getConnection()` method. (There are several other variations of the `getConnection()` method that we do not discuss at this point.)

Note that, in order to keep the example reasonably small, we have hardcoded the URL, user and password in the program. This is bad practice in a real-world program because the application has to be changed and recompiled whenever one of these values changes. To solve this 'problem', instead of using the DriverManager interface, you can use the DataSource interface. (For a more detailed discussion, see "Connecting using the DataSource API" on page 39.)

What the `getConnection()` method does is to ask each registered JDBC driver in turn whether it accepts the supplied URL. The first driver to announce that it accepts that URL will be used. For example, if you had loaded both the DB2 JDBC driver and the Oracle driver, like this:

```
Class.forName("com.ibm.db2.jcc.DB2Driver");
Class.forName("oracle.jdbc.driver.OracleDriver");
```

A URL of the form `jdbc:db2:`*`database`* will be passed to the DB2 driver, whereas a URL of the form `jdbc:oracle:`*`database`* will be passed to the Oracle driver.

When a connection can be established, the `getConnection()` method returns a Connection object (otherwise, an exception will be thrown). Using this object, you can prepare SQL statements for execution, control transactions, and query the database for metadata information describing its tables, SQL grammar, and so on.

6.1.3 Preparing an SQL statement for execution

Once the connection has been established, we send an SQL statement to the database to prepare it for execution, using the `prepareStatement()` method of the Connection object. The database checks the statement's syntax and prepares it for execution (by determining the optimal access path). If nothing went wrong, the `prepareStatement()` method in turn returns a `PreparedStatement` object.

When your statement has variable parts (in our example, the lower and upper limits for the employee's salary), you can also construct the entire statement quickly:

```
BigDecimal min, max;
...
PreparedStatement stmt =
        conn.prepareStatement(
              "SELECT LASTNAME, FIRSTNME, SALARY"
            + "  FROM DSN8710.EMP"
            + " WHERE SALARY BETWEEN " + min + " AND " + max);
```

However, for performance reasons, this is not a good idea. The database caches prepared queries so they can potentially be reused. However, the database cannot know which parts of the statement are fixed and which are variable, so the statement above would never be reused except when you supplied exactly the same values for the upper and lower bounds.

To increase reusability, SQL executed using this method can include what are called *parameter markers*. These markers are slots within the statement that represent a value that can be passed from the program to the statement on execution. Even though these values can change, the statement does not have to be prepared again.

A parameter marker is a placeholder for the variable parts of the statement. It is coded as a '?' in the SQL statement. Before the statement can be successfully executed, a value must be supplied for each parameter marker. The parameter markers are numbered continuously, beginning with 1.

6.1.4 Populating parameter markers

As explained in the previous section, we have to supply a value for each parameter marker before executing the statement. The parameter markers are set from within the code using the `setXxx()` methods of the `PreparedStatement` object. These methods have two parameters:

- The first is the number of the parameter marker that you want to set.
- The second parameter is the value that the parameter marker is to be set to.

So, the `PreparedStatement` class has several methods of the form:

```
setType(int paramIndex, Type paramValue)
```

Where *Type* is a Java data type matching the DB2 column type. For example, to set the first parameter marker, we use:

```
stmt.setBigDecimal(1, new BigDecimal(30000));
```

6.1.5 Executing the statement

Now that the statement has been prepared, and the variable parts have been populated with values, we are ready to execute the statement. Depending on the type of statement, you use either the `executeQuery()` method or the `executeUpdate()` method of the `PreparedStatement` class. The former is used with queries (that is, with SELECT statements), while the latter is used with all other types of statement (for example, INSERT, UPDATE, DELETE and statements like CREATE TABLE, which are known as DDL statements).

6.1.6 Processing the result set

The `executeQuery()` method returns a `ResultSet` object, which is used to iterate over the rows returned from a query. Typically, you process a `ResultSet` in a loop:

```
while (rs.next()) {
  // Process current row
}
```

The `next()` method is used to advance the underlying database cursor to the next row. It returns a boolean value that indicates whether or not the cursor is now positioned on a row. This means that, when calling it after the last row had been processed, it will return false, terminating the loop.

When the `ResultSet` object is positioned on a row (that is, when the `next()` method has been called and returned `true`), you use one of several ‘getter’ methods to retrieve the individual columns of the current row. There is two getter methods for each Java type that can possibly be returned as a column value: One taking an `int` parameter and one taking a `String` parameter. You can refer to a column either by its index in the list of selected columns (beginning with 1), or by its name, using the `int` or the `String` version of the getter method, respectively. In our JDBC program from Example 6-1 on page 103, we used the index:

```
String lastname = rs.getString(1);        // LASTNAME
String firstname = rs.getString(2);       // FIRSTNME
BigDecimal salary = rs.getBigDecimal(3);   // SALARY
```

Important: For maximum portability, Sun recommends that you retrieve columns from left to right (that is, starting with the lowest index), and that you retrieve each column only once per row. Some JDBC drivers have been known to produce unexpected results otherwise.

Alternatively, to refer to the columns by name, we could have coded:

```
String lastname = rs.getString("LASTNAME");
String firstname = rs.getString("FIRSTNME");
BigDecimal salary = rs.getBigDecimal("SALARY");
```

The second form is slightly slower because the JDBC runtime ultimately refers to columns by index, and has to look up the index for the column first if you referred to it by name. However, maintenance is easier since you do not need to change anything if you add or remove a column from the SELECT list.

Note that, if a column in the SELECT list is not the name of a table column but an expression (for example, the result of a scalar function such as TRIM, or of a column function such as MAX), DB2 generates a default column name (COL1, COL2, and so on). Obviously, it would be a bad idea to refer to these columns by their generated name. To override the default column name, you should use a correlation name for the column, using the AS clause:

```
PreparedStatement stmt =
  conn.prepareStatement("SELECT MAX(SALARY) AS MAXSALARY"
                      + "  FROM DSN8710.EMP");
```

Table 6-1 compares the various `getXxx()` methods of `ResultSet` with the DB2 column types, indicating whether the method can be used to retrieve a column of that type. The preferred method (or methods) are indicated in boldface.

Table 6-1 DB2 column types and ResultSet.getXxx() methods

	SMALLINT	INTEGER	DECIMAL	REAL	DOUBLE	CHAR	VARCHAR	CHAR FOR BIT DATA	VARCHAR FOR BIT DATA	DATE	TIME	TIMESTAMP	BLOB	CLOB
getByte	x	x	x	x	x	x	x							
getShort	**x**	x	x	x	x	x	x							
getInt	x	**x**	x	x	x	x	x							
getLong	x	x	x	x	x	x	x							

	SMALLINT	INTEGER	DECIMAL	REAL	DOUBLE	CHAR	VARCHAR	CHAR FOR BIT DATA	VARCHAR FOR BIT DATA	DATE	TIME	TIMESTAMP	BLOB	CLOB
getFloat	x	x	x	**x**	x	x	x							
getDouble	x	x	x	x	**x**	x	x							
getBigDecimal	x	x	**x**	x	x	x	x							
getBoolean	x	x	x	x	x	x	x							
getString	x	x	x	x	x	**x**	**x**			x	x	x		
getBytes								**x**	**x**					
getDate						x	x			**x**	x	x		
getTime						x	x			x	**x**	x		
getTimestamp						x	x			x	x	**x**		
getAsciiStream						x	x	x	x					
getUnicodeStream						x	x	x	x					
getBinaryStream								x	x					
getBlob													x	
getClob														x
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x

See also Appendix A of *Application Programming Guide and Reference for Java™*, SC26-9932, for additional information about the recommended mappings of Java data types to JDBC and SQL data types.

6.1.7 Cleaning up resources

The last step in a well-behaved application is to clean up any resources the application may have allocated. In this sample program, we could have done without proper cleanup code, since the resources will automatically be freed by the JVM and/or the operating system upon termination. However, things are very different if you are running in a 'long-living' environment such as a Web application server.

You may argue that the Java garbage collection mechanism will eventually clean up any unused objects that your program has created. However, you cannot tell when the garbage collector will eventually spring into action, and your program may well run out of resources before that point. JDBC, being an API to interface with an external system (the database), has to use and maintain external resources that are potentially limited. For example, the number of open PreparedStatements or ResultSets may be (in fact, probably *will* be) limited by the JDBC driver implementation.

To clean up the resources used by a `ResultSet`, `PreparedStatement`, or `Connection`, you call its respective `close()` method. In Example 6-1 on page 103, we call the `close()` methods in the `finally` part of a `try / catch / finally` block. The Java language specification guarantees that a `finally` block will always be executed, no matter if the code in the `try` block executed successfully or failed (threw an exception).

Annoyingly, the catch methods may themselves throw an exception, which is why we enclosed them in a try / catch block of their own, ignoring any exceptions that might have been thrown.

```
finally {
  // Clean up.
  try {
    if (rs   != null) rs.close();
    if (stmt != null) stmt.close();
    if (conn != null) conn.close();
  } catch (SQLException ignored) {
  }
}
```

Note that a `ResultSet` is automatically closed when the `PreparedStatement` that returned it is closed, and the `PreparedStatement` in turn is automatically closed when the `Connection` that created it is closed. Still, it is good practice to close everything explicitly.

6.2 Running the Hello application from WSAD

Now we are ready to run our sample application from within WSAD.

To run a program in WSAD, you create a *launch configuration*. A launch configuration contains all the information needed to invoke and run the program, such as:

- The main class name (in our case, `com.ibm.itso.sg246435.jdbc.Hello`)
- The classpath to use
- Command line arguments to the program

6.2.1 Creating the launch configuration

To create a launch configuration, select the **Hello.java** file, then click the drop-down button beside the Running Man icon. From the drop-down menu, select **Run As** → **Java Application**. WSAD will create a launch configuration for you and launch it immediately.

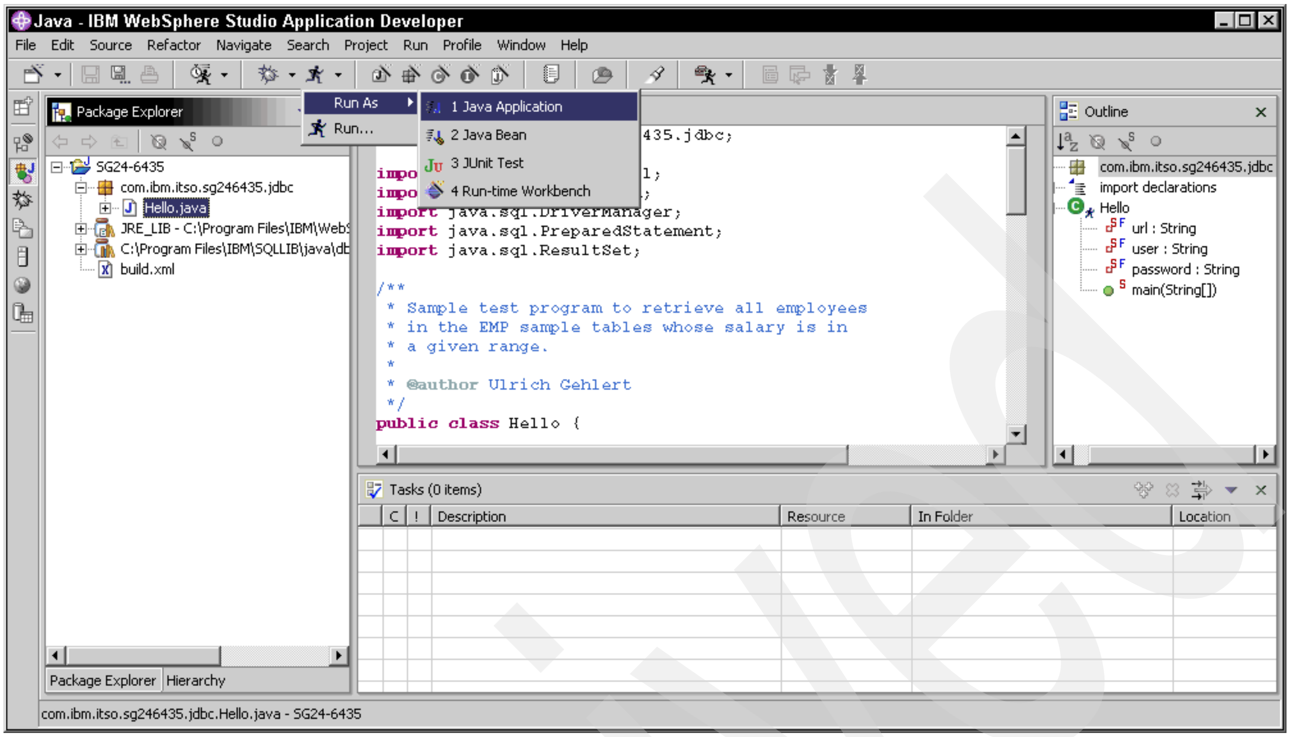

Figure 6-2 Running the application from WSAD

The output from the program goes to the Console view, which opens automatically whenever a Java program produces output by writing to System.out or System.err.

Our first attempt to run the sample program, however, produces an error message: `java.lang.ClassNotFoundException: com.ibm.db2.jcc.DB2Driver` (see Figure 6-3). This means that the Java Virtual Machine cannot find the bytecode for the indicated class, in our case, for the JDBC driver.

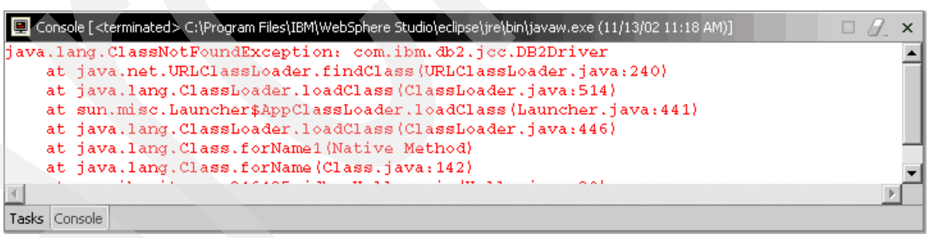

Figure 6-3 ClassNotFoundException due to missing JDBC driver

The problem is that the program requires the JDBC driver at runtime, but not at compile time. In other words, the JDBC driver implementation is not needed to compile the program (because we only use the classes and interfaces in the java.sql package, which come with the Java standard libraries), but it is certainly needed to run it.

To put additional libraries on the runtime classpath, we have to modify the launch configuration created in the previous step.

6.2.2 Setting up the classpath

To include the JDBC driver code in the runtime classpath of our launch configuration, perform the following steps:

1. Select **Run** → **Run…** (or select the drop-down beside the “Running Man” icon and select **Run…** from there). The Launch Configurations dialog opens. This dialog lists all available launch configurations (right now, there is only one of them) and allows you to create new configurations or modify existing ones.
2. Make sure that the launch configuration for the Hello program is selected, then switch to the **Classpath** tab. In the bottom half of the dialog, the Use default class path box is checked, which indicates that the runtime classpath is the same as the compile time classpath. Since we want to add a library that does not need to be on the compile time classpath, uncheck the box. Now the buttons that allow you to modify the classpath will be enabled (see Figure 6-4).

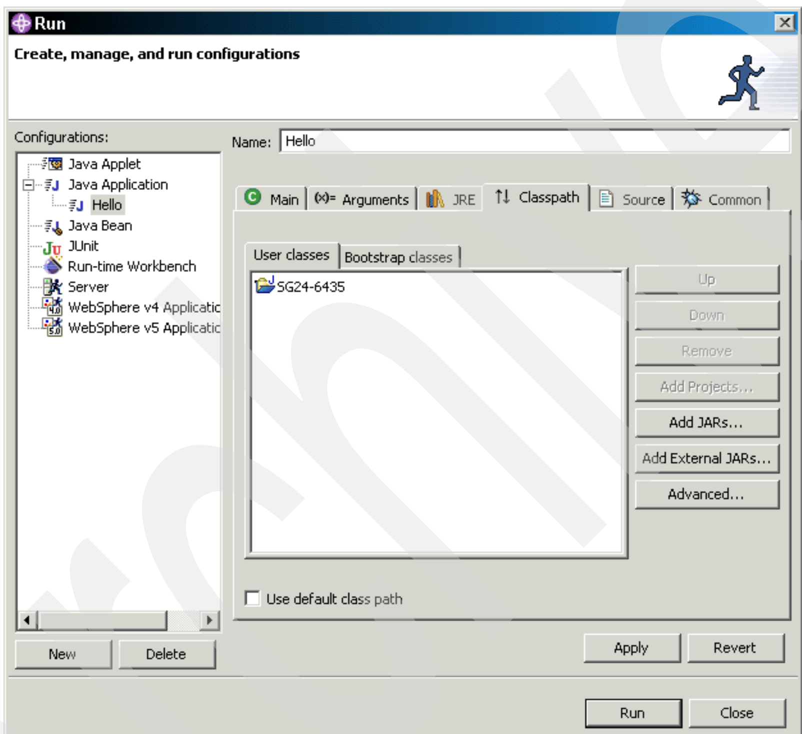

Figure 6-4 Modifying the runtime classpath in the Launch Configurations dialog

3. Press the **Add External JARs** button. In the dialog that opens, navigate to the Java subdirectory of your DB2 installation directory (by default, this will be C:\Program Files\IBM\SQLLIB\java). Select the Jar file containing the DB2 JCC driver, **db2jcc.jar**, and the Jar file named **db2jcc_license_cisuz.jar** (hold down the Ctrl key to select multiple files). Press **Open**. The Jar files have now been added to the runtime classpath.

> **Note:** The db2jcc_license_cisuz.jar file contains a license code that is necessary in order to use the DB2 Universal Driver for SQLJ and JDBC with DB2 for OS/390 and z/OS databases.

4. Click **Run** to save the modifications to the launch configuration, and re-launch the program.

If everything went smoothly, the output should look similar to Figure 6-5. Congratulations! We successfully created and ran our first program to access DB2 for z/OS from the workstation.

Figure 6-5 Output from running the Hello program

> **Tip:** To show any view in full-screen mode, double-click the view's title bar. Double-clicking the title bar again restores the previous layout.

6.2.3 Troubleshooting

If the program did not run correctly, check the stack trace that appears in the Console window. The most common errors are described in this section.

Invalid database URL syntax

Make sure the JDBC URL has the following format:

```
jdbc:db2://your.server.name:portnumber/locn
```

Where:

- *your.server.name* is your server's hos tname (or IP address).
- *portnumber* is the TCP port number on which your DB2 subsystem receives incoming SQL requests.
- *locn* is your DB2 location name, not the DB2 subsystem name (in our case both are the same, but that is usually not the case).

> **Tip:** If you do not know the port number, either ask you system administrator, issue the -DIS DDF command, or look for message DSNL519I in your DB2 system's master address space job log.

IO Exception opening socket to server

The driver could not establish a TCP/IP connection to the target database server, due to one of the following reasons:

- The port number in the JDBC URL is incorrect.
- The database server is not running.
- DDF is not active on the database server, or is not set up for TCP/IP.

Connection authorization failure occurred

The user name or password is incorrect.

An attempt was made to access a database which was not found

Make sure you specified the DB2 location name in uppercase.

FIRSTNAME IS NOT A COLUMN [...] IDENTIFIED IN A FROM CLAUSE

The column name is misspelled (in this example, it should have been be FIRSTNME). The full text message is only returned because the retrieveMessagesFromServerOnGetMessage property is enabled on the URL. Otherwise, you will see the following error information:

```
DB2 SQL error: SQLCODE: -206, SQLSTATE: 42703, SQLERRMC: FIRSTNAME
```

ILLEGAL SYMBOL

There is a syntax error in the SQL statement, for example, a misspelled statement name.

The version of the IBM Universal JDBC driver in use is not licensed for connectivity to z/OS databases. To connect to this DB2 server, please obtain a licensed copy of the IBM DB2 Universal Driver for JDBC and SQLJ.

The required license file is not on the runtime classpath. Make sure that you included db2jcc_license_cisuz.jar on the classpath.

6.3 Running the Hello application from Unix System Services

Until now, we have not been running the application on the mainframe, but on the development workstation. In this section, we deploy the code to the Unix System Services (USS) environment on z/OS and run it directly on the mainframe.

If your OS/390 or z/OS installation happens to run an SMB server such as DFS/SMB or Samba (`http://www.samba.org`), you are in luck. This enables you to access the HFS file system as a shared network drive. Otherwise, you can use FTP to transfer your code to HFS. We describe both methods in “Exporting to a shared file system” on page 113 and “Exporting via FTP” on page 115, respectively.

> **Tip:** For more information about DFS/SMB and Samba, refer to *S/390 File and Print Serving*, SG24-5330.

6.3.1 Exporting to a shared file system

To export your code to a USS directory, you first have to map that directory as a drive on your PC. From Windows Explorer, select **Tools** → **Map Network Drive**. Select a drive letter and enter the network path for the shared directory. In our case, the network path is \\wtsc63\rc63, which maps to the /u/rc63 directory on HFS (see Example 6-6 on page 114).

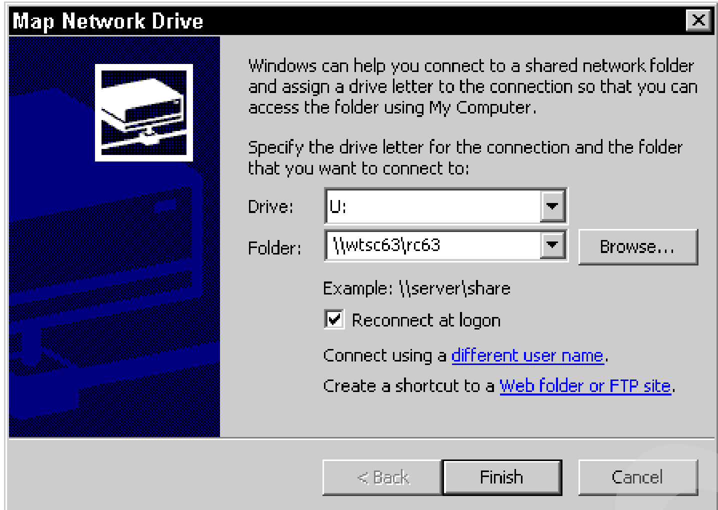

Figure 6-6 Accessing the HFS file system as a shared network drive

After mapping the network drive, you should be able to browse it in Windows Explorer just like a regular local drive.

To export the code to the mainframe, follow these steps:

1. Select the project folder (SG24-6435) then select **File** → **Export...**.
2. Select **File System** from the list of export destinations and click **Next**.
3. In the Directory field, type the drive letter to which you mapped the HFS file system, followed by a colon and a backslash. Select **Create directory structure for files** (see Figure 6-7 on page 115).
4. Click **Finish**.
5. To verify that the files have been exported correctly, use Windows Explorer to view the directory structure on the HFS file system.

> **Note:** The “Create directory structure for files” option tells WSAD to create the same directory structure on the export directory as it exists in the workspace. That is, in our example, the wizard will create a directory called SG24-6435 and create the directories corresponding to the Java package names below that directory.
>
> When you work with several projects at a time, this makes it easier to tell which file on the destination file system belongs to which project.

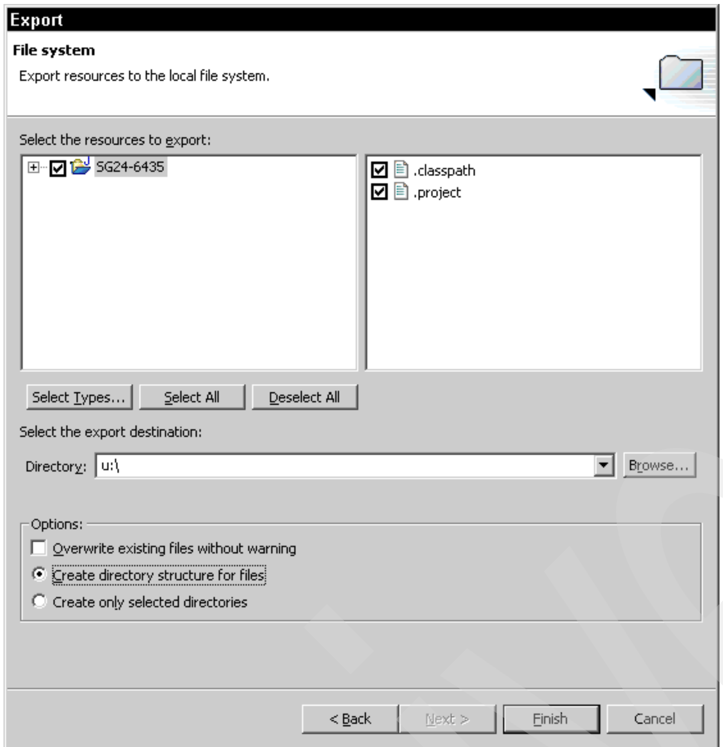

Figure 6-7 Export to file system wizard

> **Important:** Since the HFS file system on Unix System Services uses EBCDIC, the SMB server software usually performs an ASCII to EBCDIC conversion for files whose content is known to be text (such as Java source code). Make sure that it does not perform that conversion for .class files that are binary.

6.3.2 Exporting via FTP

If your installation does not offer SMB access to the HFS file system, you can use FTP to export your code. Fortunately, you do not have to do this manually; WSAD comes with built-in FTP support.

To export the code, follow these steps:

1. Select the project folder (SG24-6435) then select **File** → **Export…**.
2. Select **FTP** from the list of export destinations and click **Next**.
3. In the FTP host field, type the TCP/IP host name of your S/390 system.
4. In the FTP folder field, type the name of the target HFS directory—in our example, /u/rc63/SG24-6435 (see Figure 6-8 on page 116). Click **Next**.
5. Enter your user name and password. Click **Finish**.

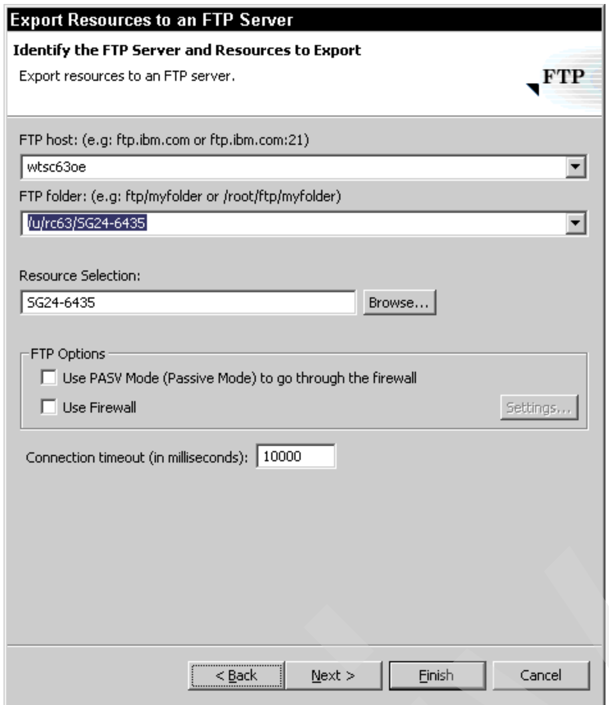

Figure 6-8 Export to FTP Server wizard

> **Note:** Unfortunately, the FTP export does not seem to allow you to do ASCII to EBCDIC translation—it always transfers files in binary. This is OK for the .class files but not if you want to ship the source code to the mainframe.

6.3.3 Running the program

Now that the program has been exported to the HFS file system, we are ready to run it from the Unix System Services command shell.

1. Log into USS via Telnet (see “Unix System Services” on page 68).
2. Change to the HFS directory to which you exported the code.
3. Make sure that your CLASSPATH environment variable contains the current directory (.) and the Jar file containing the DB2 JCC driver. To verify this, type `echo $CLASSPATH`. If not, follow the instructions in “Setting up the JDBC/SQLJ environment variables” on page 70.
4. Now, run the program by typing the command to run the Java virtual machine (**`java`**) followed by the main class (`com.ibm.itso.sg246435.jdbc.Hello`).

Figure 6-9 on page 117 shows a sample session.

```
EZYTE27I login: bartr1
EZYTE28I bartr1 Password:

Licensed Material - Property of IBM
5694-A01 (C) Copyright IBM Corp. 1993, 2001

... more legal yada yada ...

BARTR1:/u/bartr1:> cd /u/rc63/SG24-6435
BARTR1:/u/rc63/SG24-6435:> echo $CLASSPATH
.:/usr/lpp/db2/db2710/classes/db2jcc.jar
BARTR1:/u/rc63/SG24-6435:> java com.ibm.itso.sg246435.jdbc.Hello
THOMPSON, MICHAEL: $41250.00
KWAN, SALLY: $38250.00
GEYER, JOHN: $40175.00
...
```

Figure 6-9 Running the program in a USS telnet session

6.4 Running a Java program from a Windows command prompt

Another option is to run the program from a Windows command prompt. If you have WSAD there is absolutely no point in doing so, but just out of curiosity, and for those people that have to do without, we also show how to run our Hello JDBC program using commands.

6.4.1 Compile the Java program (javac)

If you have WSAD and just want to try this, you can use the same export technique we used in "Exporting via FTP" on page 115, but choose the **File system** option instead of FTP (see also "Doing it yourself - Manual program preparation for static SQLJ" on page 166) to get the data to the local file system on your machine (for example, into a directory called c:\TestJDBC\SG24-6436). To compile a Java program (and turn it into byte code) we use the **javac** command from a Windows command prompt:

```
javac com/ibm/itso/sg246435/jdbc/Hello.java
```

Make sure that you have set up your environment as described in "DB2 Universal Driver - Setup for a Windows environment" on page 71.

We have to specify the same structure defined by the Java package in our source for the program to compile correctly and turn it into a Java class.

6.4.2 Run the Java program (java)

To run a Java program in a JVM, we use the **java** command:

```
java com/ibm/itso/sg246435/jdbc/Hello
```

Note that we do not specify the extension of the program (.class) this time. The result should be identical to the execution of the same program inside WSAD.

The output of the entire process (compile, execute and result) is shown in Figure 6-10 on page 118.

```
C:\TestJDBC\SG24-6435>javac com/ibm/itso/sg246435/jdbc/Hello.java

C:\TestJDBC\SG24-6435>java com/ibm/itso/sg246435/jdbc/Hello

THOMPSON, MICHAEL: $41250.00
KWAN, SALLY: $38250.00
GEYER, JOHN: $40175.00
STIEN, IRVING: $32250.00
PULASKI, EVA: $36170.00
LUCCHESI, VINCENZO: $46500.00
HEMMINGER, DIAN: $46500.00

C:\TestJDBC\SG24-6435>
```

Figure 6-10 Compiling and running a JDBC program from the command prompt

6.5 Debugging the application on the workstation

WSAD comes with a powerful debugger that allows you to debug programs running both locally (that is, within WSAD itself) and remotely (programs that run on a JVM on another machine).

To debug a program, you first add one or more breakpoints in the code, then start the program in debug mode.

To add a breakpoint, do the following:

1. Open the source file (for example, Hello.java) where you want to set the breakpoint.
2. On the line where you want to set the breakpoint (for example, the first line of the `main()` method), place your mouse pointer in the gray bar to the left of the editor area.
3. Double-click to set the breakpoint. The breakpoint will be indicated by a marker in the gray bar.

> **Note:** Alternatively, you can check the **Stop in main** option in the launch configuration, which causes the program to stop at the first executable statement in the `main()` method when started in debug mode.

To start the program in debug mode, select the drop-down beside the Debug icon ( ) and pick the launch configuration for the program you want to debug. When the program hits the breakpoint, program execution is suspended, and WSAD switches to the Debug Perspective. The current line in the source file is highlighted. Now you can use the icons in the Debug view toolbar to control the program's execution:

- **Resume** Continues execution until the next breakpoint is hit
- **Suspend** Suspends a running program or thread
- **Terminate** Terminates a running program or thread
- **Step into** Steps into the current statement
- **Step over** Steps over the current line

-  **Step out** Steps out of the current method

6.6 Remote debugging

Both the Java Virtual Machine on z/OS and WSAD support Java Platform Debugger Architecture (JPDA), which is a Java standard for remote debugging. It allows you to connect, via TCP/IP, to a Java Virtual Machine running on a remote computer, provided that the JVM had been started with remote debugging enabled.

Figure 6-11 depicts the JPDA architecture. When the JVM starts up in debug mode, it creates a Debug listener thread. This thread listens on a TCP/IP port and receives debug control instructions, in a standard format called Java Debug Wire Protocol (JDWP), from a remote debugger connected to this port.

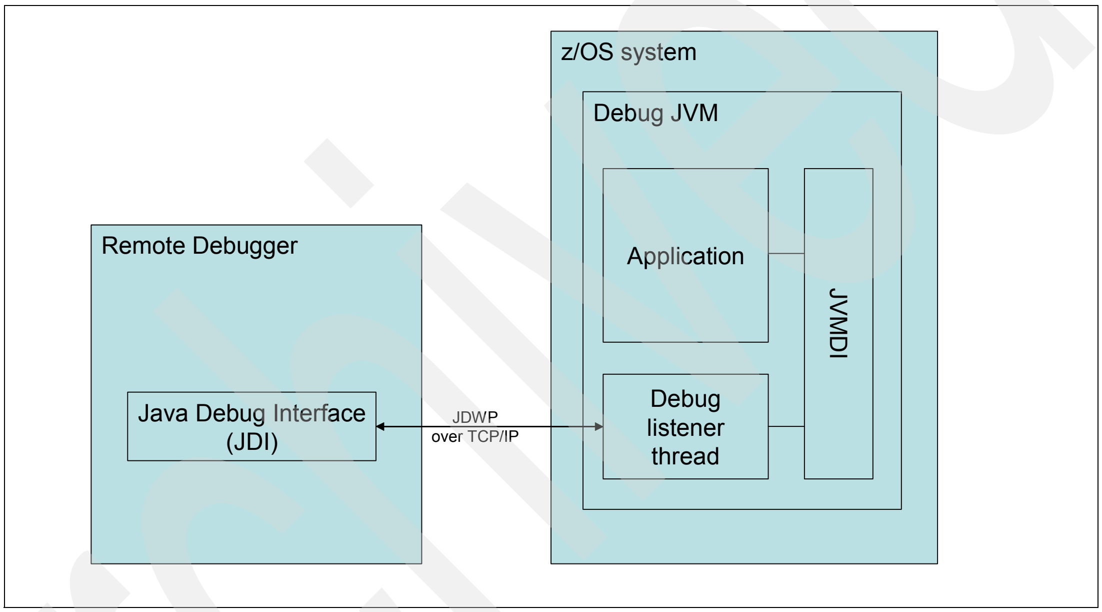

Figure 6-11 Remote debugging architecture

How exactly a JVM is enabled for remote debugging depends on the JVM implementation, and on the environment that JVM is running in (for example, CICS or WebSphere Application Server).

> **Note:** For instructions on how to set up remote debugging in WebSphere Application Server, refer to *Assembling Java™ 2 Platform, Enterprise Edition (J2EE™) Applications*, SA22-7836. For CICS applications, refer to *Java applications in CICS*, SC34-6000.

For stand-alone applications, you enable remote debugging via command line switches. For the z/OS JVM implementation, the command line looks like this:

```
java -Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=8000
  mainclass arguments
```

Where:

- suspend=y instructs the JVM to suspend after loading the main class and wait for a debugger to attach.

- 8000 is the TCP/IP port number on which you want the debug listener thread to listen. Instead of 8000, you can use any unused TCP/IP port number greater than 1024. If you are working in a team, it is probably a good idea to assign each team member a different port number to use.

> **Tip:** To get an overview of the options for remote debugging, type `java -Xrunjdwp:help`.

To remotely debug our sample program, follow these steps:

1. As described in "Running the Hello application from Unix System Services" on page 113, log into USS and change to the directory to which you exported the code.
2. Start the program in z/OS USS with the following command line:

   ```
   java -Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=8000
     com.ibm.itso.sg246435.jdbc.Hello
   ```

   The JVM will start up and then suspend, waiting for a debugger to attach to it.
3. In WSAD, select **Run** → **Debug...**. The Launch Configuration dialog opens. Select **Remote Java Application** and click **New**. In the Name field, enter a name for the new launch configuration, for example, `Remote Hello`. In the Connection Properties fields, enter the mainframe's TCP/IP host name or IP address. If you used a port number different from the default of 8000, enter that as well (see Figure 6-12).

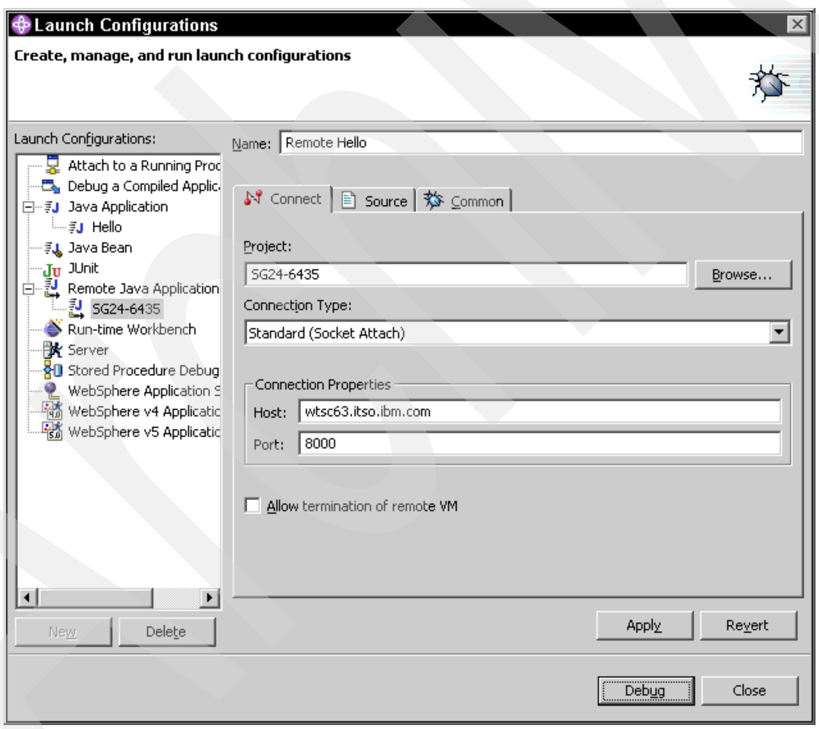

Figure 6-12 Launch configuration for remote debugging

4. Click **Debug**. WSAD switches to the Debug Perspective and attaches to the remote JVM.

Now you can debug the remote program exactly as if it was running inside WSAD on your local machine. The only thing you have to be careful about is to keep the byte code and the source code in sync, that is, to make sure that you re-export the program to the remote machine after changing the source code.

7

JDBC revisited

This chapter describes the following:

- INSERT, UPDATE and DELETE statements
- NULL handling
- Database metadata
- Positioned UPDATE and DELETE
- Large objects (LOBs)

It also demonstrates many of the APIs and techniques with a complete application, a Java version of the well-known SQL Processing Using File Input (SPUFI) tool.

7.1 INSERT, UPDATE and DELETE statements

In Chapter 6, “Getting started with JDBC” on page 101, we only covered queries that are executed by the `PreparedStatement.executeQuery()` method, returning a `ResultSet`. To execute an SQL statement that is not a query, you use the `executeUpdate()` method of `PreparedStatement`. Note that the method name may be slightly misleading—it is used not only for the SQL UPDATE statement, but for each kind of statement that does not return a `ResultSet`, including statements that create or drop objects in the database (such as CREATE TABLE and DROP TABLE) that are known as DDL statements.

In this section, we cover the three most common statements to find in an `executeUpdate()`, namely:

- INSERT to add new rows to a table
- UPDATE to change the value of columns in existing rows
- DELETE to delete rows from a table

The `executeUpdate()` method returns the number of rows affected if it executed an INSERT, UPDATE or DELETE statement, as explained in the following sections. If it executed a DDL statement (such as CREATE TABLE), it always returns zero.

7.1.1 INSERT

There are two basic forms of the INSERT statement: One to insert a single new row into a table, and one to insert several rows that have been SELECTed from another table.

The first form looks like this:

```
INSERT INTO TABLENAME (COL1, COL2, ...)
VALUES (?, ?, ...)
```

In this case, the `executeUpdate()` call always returns 1.

When you want to copy data from another table, you can use the second form of the INSERT statement:

```
INSERT INTO TARGET_TABLE (COL1, COL2, ...)
  SELECT (S1, S2, ...)
    FROM SOURCE_TABLE
```

Obviously, the number of columns in the SELECT clause must match the number of columns to be inserted, and the respective columns must have compatible types.

Here, the `executeUpdate()` method returns the number of rows inserted, that is, the number of rows that were returned from the SELECT clause.

7.1.2 UPDATE

The UPDATE statement has the basic form:

```
UPDATE TABLENAME
SET COL1 = ?
  , COL2 = ?
  , ...
WHERE CONDITION
```

The `executeUpdate()` method returns the number of rows that have been updated, that is, the number of rows that matched the condition in the WHERE clause.

7.1.3 DELETE

DELETE has the form:

```
DELETE FROM TABLENAME
 WHERE CONDITION
```

Again, the `executeUpdate()` method returns the number of rows that have been deleted (in other words, that matched the WHERE clause).

7.2 NULL handling

To indicate the absence of a value, a column in a DB2 table may contain the special value NULL (unless it has been declared NOT NULL). It is important to note that the NULL value is different from the numeric value zero, or the empty string; to say it again, it is used to record the fact that the value is unknown or not applicable.

Now Java makes a difference between primitive types, such as `short` and `int`, and object types (also called reference types) such as `String`. While there is a special null value in Java for object types, this is not the case with primitive types. When you retrieve a column that is mapped to an object type in Java, such as a CHAR column that maps to a String, there is no problem: If the column was NULL, a Java null reference will be returned.

The problem is that many of the `get`*`Xxx`*`()` methods of `ResultSet` return a value of primitive type (namely, those for database types that map to Java primitive types). The problem arises, how can you tell that the column you want to examine actually has a value, or is NULL?

> **Note:** If the column was NULL, the getXxx() methods return zero (for numeric primitive types), or false (in the case of getBoolean()). Still, you cannot know if the column's value was indeed zero, or was NULL.

There are two possible solutions. Either you can use the method `ResultSet.getObject()` which returns an object of the closest matching type. For example, when the column is a SMALLINT column, you would retrieve an instance of `java.lang.Short`, or `null` if the column was NULL.

The other possibility is to call `ResultSet.wasNull()` right after retrieving the column with, for example, `ResultSet.getShort()`. If the column was NULL, the `wasNull()` method returns `true` (and the return value from `getShort()` is zero); otherwise, it returns `false`.

7.3 Examining result sets

Suppose that you develop an interactive application that allows the user to construct a database query quickly, and you want to present the results of the query nicely formatted. In this case, you do not know the number, width, and types of the columns in the result set. This is where the ResultSetMetaData API comes into play.

It has one method taking no parameters, `getColumnCount()`, which returns the number of columns in the result set. All other methods expect as parameter a column index (starting with 1). For the indicated column, you can retrieve:

- The column's name
- The name of the table this column came from (useful if it was a JOIN of several tables)
- The column's type (one of the constants defined in java.sql.Types)

- The column's nullability (whether or not it may contain NULL values)
- Several other pieces of information about the column, for example, whether or not it is signed

We demonstrate some of these methods in "A complete example: Poor man's SPUFI" on page 130.

7.4 Database metadata

As mentioned in before, the JDBC API allows applications to retrieve database metadata. These metadata fall into three broad categories:

- Information about the JDBC driver

  Driver name, major and minor version
- Information about the database server

  SQL level supported, handling of NULL values, etc.
- Information about database objects

  Number, names and types of columns in a table

7.4.1 Information about the JDBC driver

For diagnostic purposes, you can retrieve the JDBC driver's name and version using the methods `getDriverName()`, `getDriverVersion()`, `getDriverMajorVersion()` and `getDriverMinorVersion()` in the DatabaseMetaData interface.

7.4.2 Information about the database server

The `DatabaseMetaData` interface declares a whole barrage of methods to retrieve information about various aspects of the database server to which you are currently connected. Most of these methods are of interest mainly to tool developers.

Rather than explain each and every method from this interface, we present a short demonstration program, which calls each method that takes no parameters, and prints the result of the invocation (Example 7-1). Try and run this sample against your DB2 database server.

Example 7-1 Printing database metadata

```
package com.ibm.itso.sg246435.jdbc;

import java.lang.reflect.Array;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;

/**
 * Demonstrate the database metadata API
 * by invoking all metadata methods that take no parameters
 * and printing the results.
 *
 * @author Ulrich Gehlert
 */
```

```
public class DatabaseMetaDataDemo {

  /**
   * Invokes all no-argument methods of the
   * DatabaseMetaData object and prints the result of
   * the invokation (unless it's an array).
   *
   * @param md
   *   DatabaseMetaData to be printed.
   */
  private static void printMetaData(DatabaseMetaData md) {
    Object [] noargs = { };
    Method[] methods = DatabaseMetaData.class.getMethods();

    for (int i = 0; i < methods.length; i++) {
      Method m = methods[i];
      // It's a method taking a parameter. Skip this one.
      if (m.getParameterTypes().length > 0)
        continue;

      try {
        Object result = m.invoke(md, noargs);
        // Skip methods that returned an array.
        if (result instanceof Array)
          continue;

        // Everything else should print out fine.
        System.out.println(m.getName() + ": " + result);
      } catch (Exception unexpected) {
        // Should not happen...
      }
    }
  }

  public static void main(String[] args) {
    try {
      Class.forName("com.ibm.db2.jcc.DB2Driver");
      Connection conn = DriverManager.getConnection("jdbc:db2:DB7Y");
      DatabaseMetaData md = conn.getMetaData();
      printMetaData(md);
    } catch (Exception e) {
      System.err.println(e);
    }
  }
}
```

7.4.3 Information about database objects

Suppose you want to find out the column names and types for a table. If you are a seasoned DB2 developer, you certainly know about the DB2 catalog table SYSIBM.SYSCOLUMNS which contains information about each column in each table defined in the DB2 subsystem. It is easy enough to query that table and find out the information you want.

However, this approach obviously is everything but platform independent—it ties your application to DB2. The DatabaseMetaData interface allows you to perform these catalog queries in a DBMS-independent way. Specifically, you can:

- Find all the tables whose names match certain criteria.

- For each table, in turn:
  – Enumerate the columns in that table, along with their type, length, and nullability.
  – Find the table’s primary key columns.
  – Find the table’s indices, and get statistics on the table.
  – Find privileges on the table.

Each of these methods returns a `ResultSet`, with standardized column names to retrieve the information you wanted.

We illustrate this with two examples, one directly querying the DB2 catalog (Example 7-2), and the other using the DatabaseMetaData API (Example 7-3).

Example 7-2 Find tables in a schema, querying the DB2 catalog

```
public static void db2FindTablesInSchema(Connection conn, String schemaPattern)
  throws SQLException
{
  PreparedStatement stmt = conn.prepareStatement(
      "SELECT NAME FROM SYSIBM.SYSTABLES"                                1
    + " WHERE CREATOR LIKE ?"                                            2
    + "   AND TYPE = 'T'"                                                3
    + " ORDER BY NAME");
  stmt.setString(1, schemaPattern);
  ResultSet rs = stmt.executeQuery();
  while (rs.next()) {
    System.out.println(rs.getString(1));
  }
  rs.close();
  stmt.close();
}
```

Notes on Example 7-2:

1. We want to select the name of all tables…
2. … where the schema (in DB2 terminology, the creator) matches `schemaPattern`.
3. … and where the object is a base table (as opposed to a view or alias, for example).

Obviously, this example will work only on DB2 database servers. Contrast this with Example 7-3.

Example 7-3 Find tables in a schema, using DatabaseMetaData

```
public static void jdbcFindTablesInSchema(Connection conn, String schemaPattern)
  throws SQLException
{
  DatabaseMetaData md = conn.getMetaData();
  ResultSet rs = md.getTables(null,                                      1
                              schemaPattern,                             2
                              "%",                                       3
                               new String[] { "TABLE" });                4
  while (rs.next()) {
    System.out.println(rs.getString("TABLE_NAME"));                      5
  }
  rs.close();
}
```

Notes on Example 7-3 on page 126:

1. The first parameter to `DatabaseMetaData.getTables` is a catalog name. Since DB2 for z/OS and OS/390 does not have the notion of a catalog, we can set this parameter to `null`.
2. The second parameter is the schema pattern.
3. Next comes a pattern of table names to search for. Since we want to retrieve all table names in the schema, we use the "%" wildcard, which matches any string.
4. The last parameter designates a list of table types to be returned (for example, this could include views, aliases, or other special table types supported by the database server). The possible values are DBMS-dependent; the `DatabaseMetaData.getTableTypes()` method returns a list of all available table types.
5. The `getTables()` method returns a `ResultSet` with standardized column names, including `TABLE_NAME`. For the list of column names, refer to the JDBC API.
   Also, the JDBC API specifies the ordering of result set rows. In the case of `getTables()`, the rows are ordered by table type, table schema and table name.

> **Important:** The catalog metadata API requires the catalog stored procedures to be installed on the database server. See "Required DB2 for z/OS changes to enable the Universal Driver" on page 58 for more details.

7.5 Positioned UPDATE and DELETE

In "INSERT, UPDATE and DELETE statements" on page 122, we only covered *searched* UPDATE and DELETE statements; that is, the rows that are affected by the UPDATE or DELETE are those matching the predicate in the WHERE clause.

Using positioned UPDATE and DELETE, it is possible to change the underlying table on-the-fly while iterating through the result set. This is useful in situations where the criteria for rows to be updated cannot be expressed in SQL, for example, because they depend on interactive user input.

By default, result sets are not updateable. To create an updateable result set, use the three-argument form of Connection.prepareStatement:

```
PreparedStatement prepareStatement(String sql,
                                   int resultSetType,
                                    int resultSetConcurrency)
```

Where `resultSetConcurrency` is one of:

- `ResultSet.CONCUR_READ_ONLY`, resulting in a non-updateable cursor (the default), or
- `ResultSet.CONCUR_UPDATABLE`, resulting in an updateable cursor.

The second parameter, `resultSetType`, determines whether you get a forward-only cursor or a scrollable cursor (see "Scrollable cursors" on page 129). The default is `ResultSet.TYPE_FORWARD_ONLY`.

7.5.1 Positioned UPDATE

To update the row a ResultSet is currently positioned on, you call one of the updateXxx() methods of class ResultSet. Like the corresponding getXxx() methods, you can use either a column index or a column name to refer to the column.

To set a column to a NULL value, call the updateNull() method, or one of the updateXxx() methods taking a reference type argument with a null argument.

However, the updateXxx() methods do not update the database row immediately; rather, the changes are only recorded in the ResultSet object. To actually change the database, call the updateRow() method.

Example 7-4 Using updatable cursors

```
private static void increaseSalary(BigDecimal amount) throws SQLException {
  PreparedStatement stmt =
    conn.prepareStatement("SELECT SALARY FROM DSN8810.EMP",
                          ResultSet.TYPE_SCROLL_SENSITIVE,
                          ResultSet.CONCUR_UPDATABLE);
  try {
    ResultSet rs = stmt.executeQuery();
    try {
      while (rs.next()) {
        BigDecimal oldSalary = rs.getBigDecimal("SALARY");
        rs.updateBigDecimal("SALARY", oldSalary.add(amount));
        rs.updateRow();
      }
    } finally {
      rs.close();
    }
  } finally {
    stmt.close();
  }
}
```

Restriction: Positioned UPDATE is only supported with DB2 Version 8 and later. We tested the example on a pre-release version of DB2 Version 8 for z/OS.

7.5.2 Positioned DELETE

Positioned DELETE is easy: To delete the row the ResultSet is currently positioned on, call ResultSet.deleteRow(). The difference between positioned UPDATE and positioned DELETE is that deleteRow() takes effect immediately, while the updateXxx() methods take effect only when updateRow() is called.

7.6 Large objects (LOBs)

LOBs allows applications to store large binary or text data, such as images, word processor documents, or large XML files.

Basically, there are two different ways to handle LOB columns from a Java application. The easy way is to read or write the LOB data in one large chunk, using the getBytes() or setBytes() methods, respectively, for BLOB data, or the get/setString() methods for CLOB data.

However, this is not a good idea if the LOB data get very large (which, after all, is the point of using LOBs) since this uses a lot of memory. Maybe you only want to figure out how long a LOB column's value is; in this situation, it is an obvious waste of resources to read the entire LOB data only in order to count the number of bytes or characters.

The alternative is to either use Java streams or the java.sql.Blob and java.sql.Clob interfaces. Table 7-1summarizes the different JDBC methods to work with LOB data. Note that you cannot construct a BLOB or CLOB object yourself (they are Java interfaces, not regular Java classes); the JDBC runtime does that for you when retrieving LOB data. Therefore, the setBlob() and setClob() methods are useful only for copying the value of a LOB column from one row or column to another one.

Table 7-1 JDBC methods for use with LOB types

Operation	Using simple type	Using LOB type
	BLOBs	
SELECT	ResultSet.getBytes()	ResultSet.getBlob() ResultSet.getBinaryStream()
INSERT, UPDATE	PreparedStatement.setBytes()	PreparedStatement.setBinaryStream() PreparedStatement.setBlob()
	CLOBs	
SELECT	ResultSet.getString()	ResultSet.getClob() ResultSet.getCharacterStream()
INSERT, UPDATE	PreparedStatement.setString()	PreparedStatement.setCharacterStream() PreparedStatement.setClob()

7.7 Scrollable cursors

Scrollable cursors, a feature introduced in JDBC 2.0, allow you to move through a result set backward as well as forward, and to position the cursor to an absolute or relative position within the result set.

To request a scrollable cursor, use the three-argument form of `Connection.prepareStatement`:

```
PreparedStatement prepareStatement(String sql,
                                   int resultSetType,
                                    int resultSetConcurrency)
```

Where:

- `resultSetType` is one of `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE` or `TYPE_SCROLL_SENSITIVE`
- `resultSetConcurrency` is one of `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`, as explained in "Positioned UPDATE and DELETE" on page 127

All of the constants are defined in class `ResultSet`.

The `resultSetType` argument determines if the result set is scrollable, and if so, if it is aware of changes to the underlying table:

- `TYPE_FORWARD_ONLY` results in a non-scrollable result set, which is the default if you use the one-argument or two-argument form of `prepareStatement()`.
- `TYPE_SCROLL_INSENSITIVE` results in a scrollable result set which is not aware of changes to the underlying result table.
- `TYPE_SCROLL_SENSITIVE` results in a scrollable result set which is aware of changes to the underlying result table.

Note: Be careful to get the parameter order right: the result set type is the second parameter, the result set concurrency is the third parameter. Since both are of type `int`, the Java compiler will not complain if you got it the wrong way round.

Restriction: The combination of `TYPE_SCROLL_INSENSITIVE` with `CONCUR_UPDATABLE` is not supported with the DB2 Universal Client.

7.8 A complete example: Poor man's SPUFI

In this section, we demonstrate some techniques presented in the previous sections with a complete example program—a Java version of the well-known SPUFI application (SQL Processing Using File Input).

The SPUFI tool that comes with DB2 for z/OS and OS/390 reads SQL statements from a sequential file, processes those statements, and displays the results in an ISPF browse session. Our Java version does a similar job, except that it can process multiple files (or, in general, Java input streams which may or may not be files on disk), and can print the results to an arbitrary output stream, by default, System.out.

The class declaration and constructors

We start by showing the class declaration and constructors (Example 7-5).

Example 7-5 Spufi class declaration and constructors

```
package com.ibm.itso.sg246435.jdbc;

import gnu.getopt.Getopt;
import gnu.getopt.Getopt.LongOpt;

import java.io.*;
import java.sql.*;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/**
 * A poor man's SPUFI (SQL Processing Using File Input).
 *
 * Processes SQL statements from one or more input streams
 * (such as files), and displays the results.
 *
 * @author Ulrich Gehlert
 */
public class Spufi {

  /** Maximum display width of a column. */
  public static final int MAXCOLWIDTH = 256;

  /** Default JDBC driver name. */
  public static final String DEFAULT_DRIVER = "com.ibm.db2.jcc.DB2Driver";

  /** JDBC connection. */
  private final Connection conn;                                          1

  /** Output stream. */
```

```
  private final PrintWriter out;                                             2

  /** Error stream. */
  private final PrintWriter err;                                             3

  /** Automatically commit after each statement? */
  private boolean autoCommit;                                                4

  /**
   * Buffer for reading SQL statements and printing rows;
   * allocated as an instance variable to reduce garbage collection.
   */
  private StringBuffer buf = new StringBuffer(1000);

  /**
   * Contains Readers from which to read SQL statements.
   */
  private List streams = new ArrayList();                                    5

  /**
   * Construct a new Spufi which logs to System.out
   * and System.err.
   *
   * @param conn
   *   JDBC connection to use.
   */
  public Spufi(Connection conn) {                                            6
    this(conn, new PrintWriter(System.out), new PrintWriter(System.err));
  }

  /**
   * Construct a new Spufi which logs to the specified
   * output and error streams.
   *
   * @param conn
   *   JDBC connection to use.
   * @param out
   *   Output stream.
   * @param err
   *   Error stream.
   */
  public Spufi(Connection conn, PrintWriter out, PrintWriter err) {          7
    this.conn = conn;
    this.out = out;
    this.err = err;
  }
  ...
}
```

Notes on Example 7-5 on page 130:

1. The JDBC connection to be used
2. Any normal output, such as result sets, goes to this stream…
3. … while any error output (when an exception was thrown) goes to this one
4. If this flag is true, commit automatically after each statement
5. The list of input streams from which SQL statement are read in turn

6. This constructor uses System.out and System.err, respectively, as output and error streams
7. This constructor allows output and error messages to go to an arbitrary stream

Adding input streams

Next, we implement methods to add an input source for SQL statements (Example 7-6). In the most general form, this can be a Java text input stream (a Reader); for convenience, there are two additional methods which take the name of a file and a single SQL statement string, respectively. Each method adds the input source to the list of streams to be processed.

Example 7-6 Spufi.addStream(), addFile(), and addSql()

```
/**
 * Add a stream to read SQL statements from.
 *
 * @param reader
 *   Stream to add.
 */
public void addStream(Reader reader) {
  streams.add(reader);
}

/**
 * Add a file to read SQL statements from.
 *
 * @param filename
 *   Name of the file.
 * @throws FileNotFoundException
 *   File could not be found.
 */
public void addFile(String filename) throws FileNotFoundException {
  addStream(new FileReader(filename));
}

/**
 * Add a String to read SQL statements from.
 *
 * @param sql
 *   SQL text.
 */
public void addSql(String sql) {
  addStream(new StringReader(sql));
}
```

Executing statements from each input stream

The execute() method processes all streams which had been added using one of the methods shown in Example 7-6 in sequence. If an exception occurs while reading from a stream, it logs the error and continues with the next stream.

Example 7-7 Spufi.execute()

```
/**
 * Read all input streams in sequence and execute SQL statements.
 *
 * @throws SQLException
 */
public void execute() {
  Iterator current = streams.iterator();
```

```
  while (current.hasNext()) {                                    1
    Reader reader = (Reader) current.next();
    try {
      execute(reader);                                           2
    } catch (IOException e) {
      // Exception reading from the current stream.
      // Log and continue with next stream.
      logError(e);                                               3
    } finally {
      try {
        reader.close();
      } catch (IOException ignored) {
      }
    }
  }
}
```

Notes on Example 7-7 on page 132:

1. Loop over all input streams that had been added using addStream(), addFile(), or addSql()
2. Reads and executes SQL statements from the current stream until the end of the stream has been reached
3. Log any error that occurred while reading the stream

The execute(Reader) method is not shown here; it merely reads SQL statements (terminated by semicolons) from the stream, removing comment lines, and calls execute(String) for each line read. It terminates when the end of the stream has been reached.

Executing a single SQL statement

The execute(String sql) method takes one SQL statement and prepares and executes it. It has to determine whether the statement is a query or not; in the case of a query, it prints the result set and the number of rows found, otherwise it prints the number of rows affected by the statement (for example, the number of rows deleted if it was a DELETE statement).

Example 7-8 Spufi.execute()

```
/**
 * Execute a single SQL statement.
 * The statement and results are logged to the output streams,
 * as are any exceptions resulting from execution.
 *
 * @param sql
 *   SQL statement to execute.
 */
private void execute(String sql) {
  sql = sql.trim();                                              1
  if (sql.length() == 0)
    return;
  PreparedStatement stmt = null;
  try {
    stmt = conn.prepareStatement(sql);                           2

    int numRows;
    if (isSelectStatement(sql)) {                                3
      numRows = executeQuery(stmt);                              4
      log(numRows + " row(s) selected.");
    } else {
      numRows = executeUpdate(stmt);                             5
```

```
      log(numRows + " row(s) affected.");
    }
    if (isAutoCommit())                                              6
      conn.commit();
  } catch (SQLException e) {
    logError(e);                                                     7
  } finally {
    close(stmt);                                                     8
  }
}
```

Notes on Example 7-8 on page 133:

1. Remove leading and trailing blanks; return to the caller if the statement was empty.
2. Prepare the statement. If the statement had a syntax or authorization error, or if any other database problem was encountered, the catch block will be executed.
3. If it's a SELECT statement…
4. … call executeQuery() which displays the result set and returns the number of rows found, which we log to the output stream.
5. Else, call executeUpdate(), and log the number of rows affected to the output stream.
6. If auto commit is turned on, commit the current transaction.
7. If an error occurred, log it to the error stream. The exception is not thrown again, so our caller can happily continue with the next SQL statement.
8. Clean up, using a utility method (not shown here; see the complete source code in the Appendix).

The executeUpdate() method is easy—it simply calls stmt.executeUpdate() and returns the number of rows affected (Example 7-9). Recall that PreparedStatement.executeUpdate() returns the number of rows affected in the case of an INSERT, UPDATE, or DELETE statement, and zero for DDL statements such as CREATE TABLE.

Example 7-9 Spufi.executeUpdate()

```
/**
 * Execute an update (that is, non-SELECT) statement.
 *
 * @param stmt
 *   Statement to execute.
 * @return
 *   Number of rows affected.
 * @throws SQLException
 */
private int executeUpdate(PreparedStatement stmt) throws SQLException {
  return stmt.executeUpdate();
}
```

Displaying result sets

The executeQuery() method is more interesting. It first executes the database query, receiving a ResultSet. Then it prints a column header, showing the column names. Next, it processes the result set one row at a time, printing the row data and counting the number of rows retrieved (Example 7-10).

Example 7-10 Spufi.executeQuery()

```
/**
```

```
 * Execute a SELECT statement, printing the result set.
 *
 * @param stmt
 *   SELECT statement to execute.
 * @return
 *   Number of rows selected.
 * @throws SQLException
 */
private int executeQuery(PreparedStatement stmt) throws SQLException {
  ResultSet rs = null;

  try {
    rs = stmt.executeQuery();                                          1
    int[] columnWidths = getColumnWidths(rs.getMetaData());            2
    printHeader(rs.getMetaData(), columnWidths);                       3
    int rowcount = 0;
    while (rs.next()) {                                                4
      printRow(rs, columnWidths);                                      5
      rowcount++;                                                      6
    }
    return rowcount;                                                   7
  } finally {
    if (rs != null)                                                    8
      rs.close();
  }
}
```

Notes on Example 7-10 on page 134:

1. Execute the query.
2. Determine preferred column widths to be used for the header and data rows.
3. Print the header row, showing the column names.
4. While there are more rows…
5. … print each row in turn (see Example 7-12 on page 136).
6. … and count the number of rows retrieved.
7. Return the number of rows retrieved.
8. Clean up. Note that here, we use a try / finally block to clean up before the method terminates (no matter if it terminates normally, returning the row count, or abnormally, throwing an exception). Although the finally block, syntactically, comes after the return statement, it is executed before the method actually returns to the caller.

A helper method, getColumnWidths(), computes the preferred display width of the columns in the result set (Example 7-11). For each column, it takes the maximum of the column width and the column name, adding 1 for a blank character to separate the columns. It uses two methods of the ResultSetMetaData API, namely, getColumnLabel() and getColumnDisplaySize().

Example 7-11 Spufi.getColumnWidths()

```
/**
 * Determine column display widths.
 *
 * @param md
 *   Result set meta data.
 * @return
 *   Array containing the column display widths.
```

```
 * @throws SQLException
 */
private int[] getColumnWidths(ResultSetMetaData md) throws SQLException {
  final int columnCount = md.getColumnCount();
  int[] columnWidths = new int[columnCount];
  for (int col = 0; col < columnCount; col++) {
    int colWidth =
      1 + Math.max(
            md.getColumnLabel(col + 1).length(),
            md.getColumnDisplaySize(col + 1));
    if (colWidth > MAXCOLWIDTH)
      colWidth = MAXCOLWIDTH;
    columnWidths[col] = colWidth;
  }
  // Last column does not need a blank appended
  columnWidths[columnCount - 1]--;
  return columnWidths;
}
```

Notes on Example 7-11 on page 135:

1. Allocate an array of integers, one for each column
2. The column's display width is one plus either…
3. … the length of the column name
4. … or the width of the column, whichever is bigger
5. However, we truncate at MAXCOLWIDTH characters

The printRow() method prints a single row from the result set, calling the getColumnValue() method for each column, and padding the output with blanks (Example 7-12).

Example 7-12 Spufi.printRow()

```
/**
 * Prints a single row of the result set.
 *
 * @param rs
 *   Result set to process.
 * @param columnWidths
 *   Array of column display widths.
 * @throws SQLException
 */
private void printRow(ResultSet rs, int[] columnWidths) throws SQLException {
  buf.setLength(0);   // Clear buffer.
  int buflen = 0;
  final int columnCount = rs.getMetaData().getColumnCount();
  for (int col = 1; col <= columnCount; col++) {
    buflen += columnWidths[col - 1];
    buf.append(getColumnValue(rs, col));
    // Truncate buffer if too long (a column's value was longer than MAXCOLWIDTH).
    if (buf.length() > buflen)
      buf.setLength(buflen);
    while (buf.length() < buflen) // Pad with blanks.
      buf.append(' ');
  }
  log(buf.toString());            // Print current row.
}
```

Finally, the getColumnValue() method returns an object representation of a column, or null if a NULL value was encountered. LOB columns get special treatment because we do not want to print or even retrieve the entire LOB column but rather only the first few bytes or characters.

Example 7-13   Spufi.getColumnValue()

```
/**
 * Get a column's value. BLOB and CLOB columns are treated
 * separately; all other column types use the default
 * representation.
 *
 * @param rs
 *   ResultSet to process.
 * @param col
 *   Column index.
 * @return
 *   The column's value (may be <code>null</code>
 *   if tne column was NULL).
 *
 * @throws SQLException
 */
private Object getColumnValue(ResultSet rs, int col) throws SQLException {
  switch (rs.getMetaData().getColumnType(col)) {                        1
    case Types.BLOB:
      // Convert BLOB data to a hex string.
      Blob blob = rs.getBlob(col);
      if (blob == null)
        return null;
      else {
        byte[] blobBuf = blob.getBytes(0, MAXCOLWIDTH/2);               2
        return toHexString(blobBuf);
      }
    case Types.CLOB:
      // Get only the first MAXCOLWIDTH characters of the CLOB column.
      Clob clob = rs.getClob(col);
      if (clob == null)
        return null;
      else
        return clob.getSubString(0, MAXCOLWIDTH);                       3
    default:
      // All other column types.
      return rs.getObject(col);                                         4
  }
}
```

Notes on Example 7-13:

1. Examine the column type. BLOB and CLOB columns get special treatment.
2. If it was a non-NULL BLOB column, read the first MAXCOLWIDTH/2 bytes of the BLOB and return a hex string representation of the data. We divide by two since each byte needs two characters in hex format.
3. If if was a non-NULL CLOB column, return the first MAXCOLWIDTH characters.
4. For all other column types, simply return the object representation of the column value. For example, if it was an INTEGER column, this will return an instance of java.lang.Integer, or null if the column was NULL.

The main method

Finally, we show the main method which evaluates the command line arguments, sets up the input streams, and starts execution of SQL statements (Example 7-14). It uses several helper methods to parse the command line, load the JDBC driver, and display usage information.

Example 7-14 Spufi.main()

```
public static void main(String[] args) {
  try {
    int optind = evalOptions(args);                                          1
    loadDriver();                                                            2
    Connection conn = openConnection();
    conn.setTransactionIsolation(isolationLevel);

    Spufi spufi = new Spufi(conn);                                           3

    // Add SQL statements supplied from the -s command line option.
    if (sql.length() > 0)
      spufi.addSql(sql);

    // No filename arguments given?
    if (optind == args.length) {
      if (sql.length() == 0) {
        // No command-line SQL statements given as well.
        // Read SQL statements from standard input.
        spufi.addStream(new InputStreamReader(System.in));
      } else {
        // Only command-line SQL statements.
        // Turn auto commit on.
        autoCommitSwitch = true;
      }
    }

    spufi.setAutoCommit(autoCommitSwitch);

    // Add files to process.
    while (optind < args.length) {                                           4
      spufi.addFile(args[optind++]);
    }

    // Ready to go.
    spufi.execute();

    conn.close();
  } catch (Exception e) {
    e.printStackTrace(System.err);
  }
}
```

Notes on Example 7-14:

1. Evaluate the command line options, storing relevant information in static variables. Remember the index of the first non-option item on the command line.
   To parse the command line, we use the GNU getopt package (not shown here).
2. Load the JDBC driver, open the connection, and set the isolation level (the default level may have been overridden by a command line switch).
3. Create a Spufi instance, passing the JDBC connection.

4. All remaining arguments on the command line, if any, are file names. Add the files to the Spufi object to process.

To obtain the full “poor man’s SPUFI program”, see Appendix C, “Additional material” on page 313.

8

Getting started with SQLJ

In one of the previous chapters we developed a simple JDBC program to list the contents of a DB2 table. In this chapter we develop the same program using SQLJ, instead of JDBC.

Again, we run the application first from the development workstation, then on the z/OS machine.

Note that, in order to keep things simple, we only show the translation part of the SQLJ program preparation process, meaning that the application runs uncustomized, and therefore uses dynamic SQL behind the scenes. Therefore, do not expect the application to run faster than its JDBC equivalent.

8.1 Creating the source file

Our sample SQLJ program will reside in the same project as the JDBC version developed earlier in this publication. To keep things nicely separated, we create a different Java package for the SQLJ version.

Note: If you are using the WSAD team support (or plan to do so later), it is a good idea to tell WSAD to treat files ending with the .sqlj extension as text (not binary) files. Select **Window** → **Preferences**, expand the **Team** subtree (CSV - File Content), and add `sqlj` to the list of file extensions.

To create the package and the SQLJ source file, do the following:

1. In the Java perspective, select the project then click the **New Java Package** icon.
2. Enter the package name, `com.ibm.itso.sg246435.sqlj`, and click **Finish**.
3. Now we create the SQLJ source file for our Hello program. Select the package, then select **File** → **New** → **Other** → **Data** → **SQLJ** → **SQLJ File** (Figure 8-1), and click **Next**.

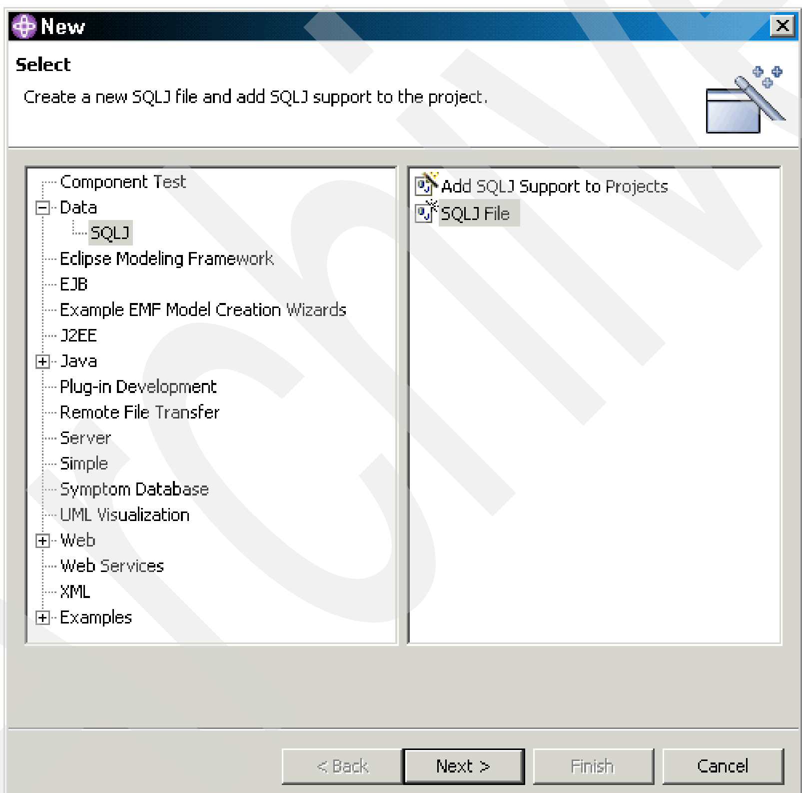

Figure 8-1 Create a new SQLJ file

In the dialog that appears (Figure 8-2 on page 143), enter `Hello` in the Name field.

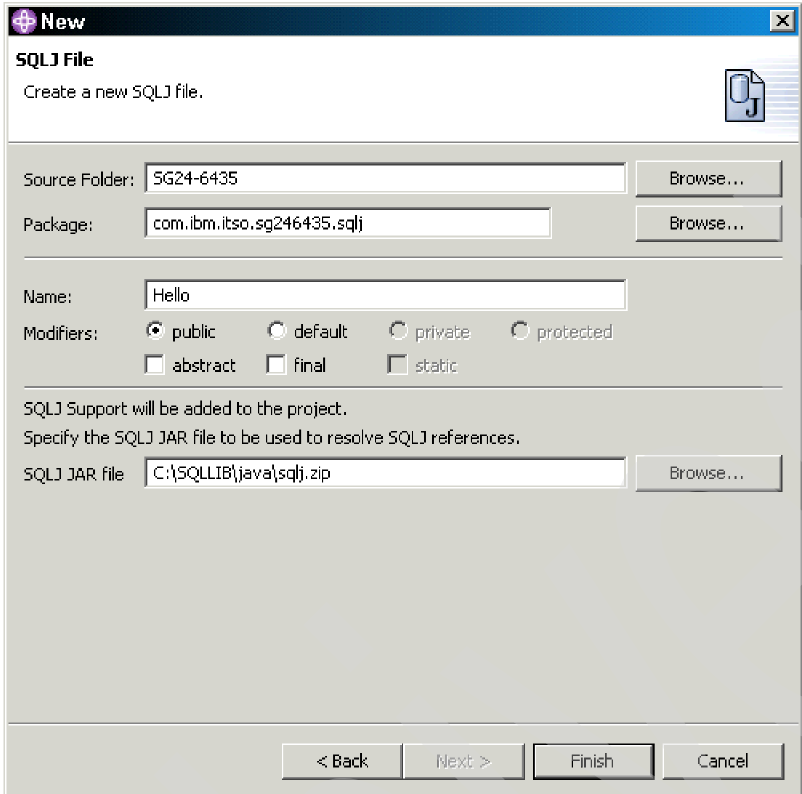

Figure 8-2 Create a new SQLJ file - 2

Then click **Finish**.

4. In the editor window that opens, type the source code for our Hello application, as shown in Example 8-1.

Do not care too much about the SQLJ language constructs for now. We describe them in much more detail in Chapter 10, "SQLJ tutorial and reference" on page 175.

Example 8-1 Source code for the Hello application

```
package com.ibm.itso.sg246435.sqlj;

import java.math.BigDecimal;

import java.sql.SQLException;

/**
 * Sample test program to retrieve all employees
 * in the EMP sample tables whose salary is in
 * a given range.
 *
 * @author Ulrich Gehlert
 */

public class Hello {

  #sql public static context Ctx;                                       1

  #sql public static iterator EmployeeIterator (String, String, BigDecimal); 2
```

```
/**
 * Load JDBC driver and initialize SQLJ connection context.
 *
 * @return
 *   The SQLJ connection context.
 */
private static Ctx initialize(String url, String user, String password)
  throws ClassNotFoundException, SQLException
{
  Class.forName("com.ibm.db2.jcc.DB2Driver");                              3
  return new Ctx(url, user, password, false);                              4
}

public static void main(String[] args) {
  if (args.length != 5) {                                                  5
    System.err.println(
      "Usage: java "
        + Hello.class.getName()
        + " <url> <user> <password> <min> <max>");
    return;
  }
  BigDecimal min = new BigDecimal(args[3]);
  BigDecimal max = new BigDecimal(args[4]);

  Ctx ctx = null;                                                          6
  EmployeeIterator iter = null;                                            7
  try {
    ctx = initialize(args[0], args[1], args[2]);                           8
    #sql [ctx] iter = {                                                    9
        SELECT LASTNAME
             , FIRSTNME
             , SALARY
          FROM DSN8710.EMP
         WHERE SALARY BETWEEN :min AND :max
         ORDER BY LASTNAME, FIRSTNME
    };
    String lastname = null;                                                10
    String firstname = null;
    BigDecimal salary = null;
    while (true) {
      #sql {
        FETCH :iter                                                        11
         INTO :lastname
            , :firstname
            , :salary
      };
      if (iter.endFetch())                                                 12
        break;
      System.out.println(lastname + ", " + firstname + ": $" + salary);
    }
  } catch (Throwable e) {
    e.printStackTrace();
  } finally {
    try {                                                                  13
      if (iter != null) iter.close();
      if (ctx != null) ctx.close();
    } catch (SQLException ignored) {
    }
  }
}
```

```
}
```

Notes on Example 8-1 on page 143:

1. Declares a connection context class for the SQL statements in the program.
2. Declares a positioned iterator for the SELECT statement at step 9.
3. Loads the JDBC driver.
4. Creates and initializes the connection context, and returns it. For information about connection contexts, see “Connection contexts” on page 188. For now, think of a connection context as representing a database connection, much like a Connection object in JDBC.
5. Unlike in the previous chapter, we do not hard code the connection information (URL, user name and password), but expect it as command line arguments. The lower and upper limits for the salary are passed as command line arguments as well.
6. Declare a connection context instance.
7. Declare an iterator variable to hold the result of the query.
8. Call the initialization method, passing the required connection information, and storing the connection context instance into the variable declared in step 6.
9. Perform the database query and assign the result to the iterator variable declared in step 7. The query runs under the connection context obtained in step 8.
10. Declares Java variables to receive the column values. For technical reasons, the variables must also be initialized; otherwise, the Java compiler will complain about using a potentially uninitialized variable.
11. Populates the variables declared in 10 with the values from the current row.
12. Terminates the loop if no more rows were found.
13. In the final block, clean up resources by closing the iterator and the context.

As soon as you save the file, WSAD automatically invokes the SQLJ translator, which then produces both the translated Java file (which, in turn, is automatically compiled to bytecode by WSAD), and a *serialized profile* (Hello_SJProfile0.ser). The SQL translator is discussed in detail in Chapter 9, “The SQLJ program preparation process” on page 149.

8.2 Running the Hello application from WSAD

To run the SQLJ sample, we create a new launch configuration as we did in “Running the Hello application from WSAD” on page 109. Then we modify the launch configuration to specify command line parameters.

8.2.1 Creating the launch configuration

To create a launch configuration, select the **Hello.java** file (which was generated by the SQLJ translator in the previous step, and not the Hello.sqlj file), then press the drop-down button beside the “Running Man” icon. From the drop-down menu, select **Run As -> Java Application**. WSAD will create a launch configuration called Hello (1) and launch it.

Since the program expects command line arguments, but we did not provide any, it just prints an error message and exits (see Figure 8-3 on page 146).

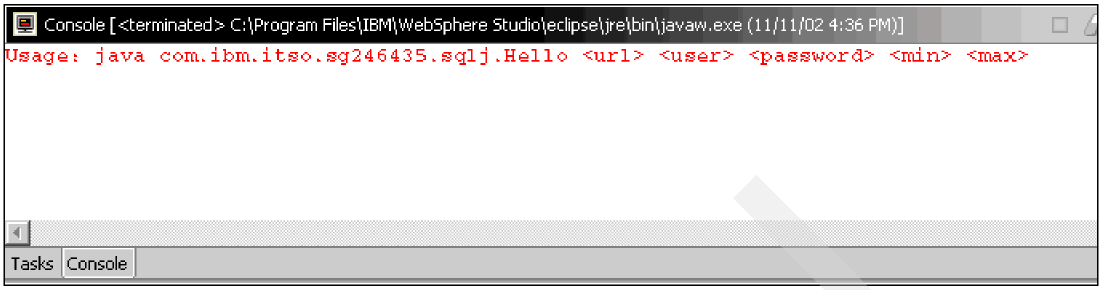

Figure 8-3 Error message from running the Hello program

8.2.2 Specifying command line parameters

To specify the command line parameters, we have to modify the run configuration we created in the previous step. Open the Launch Configurations dialog by selecting **Run -> Run…** (or select the drop-down beside the Running Man icon and select **Run…** from there). The Launch Configurations dialog opens. This dialog lists all available launch configurations (see also “Setting up the classpath” on page 111). Make sure that the launch configuration for the SQLJ program called Hello (1) is selected. This is a good time to rename the configuration to, say, Hello SQLJ. Now switch to the **Arguments** tab. In the Program arguments text box, type the five arguments our program expects, separated by blanks (see Figure 8-4).

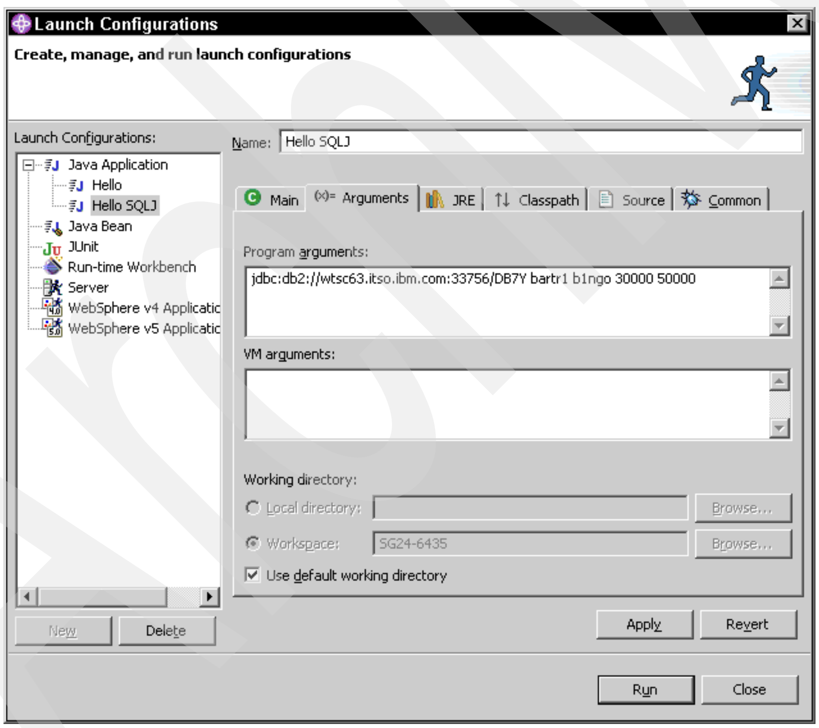

Figure 8-4 Launch Configurations dialog

Also make sure that the CLASSPATH is set up correctly (including the db2jcc.jar and db2jcc_license_cisuz.jar files).

Click **Run** to save the changes, to launch the configuration, and to run the program again. Now the output should look similar to Figure 8-5 on page 147.

Figure 8-5 Output from running the Hello program

8.3 Running the Hello application from Unix System Services

To run the application from Unix System Services, follow the same steps as in the JDBC exampe in “Running the Hello application from Unix System Services” on page 113. If you use a network file system to export the code, make sure that your SMB server does *not* perform an ASCII to EBCDIC conversion for the .ser files. They are binary files and should not be translated.

8.4 Running the Hello application from MVS batch

Occasionally, you may want to run your Java application in MVS batch, for example, if it is part of a larger job. This is what the BPXBATCH utility program is for (in fact, BPXBATCH can be used to run any USS application or command script). For more information about BPXBATCH, refer to *Unix System Services Command Reference*, SA22-7802.

Unfortunately, the standard output and standard error (in Java, System.out and System.err, respectively) cannot go to a MVS or JES data set; they must be HFS files. In the sample JCL (see Example 8-2 below), we call the TSO ocopy program to copy these to a JES data set.

Example 8-2 Sample JCL to run a Java application in MVS batch

```
//BARTJAVA JOB (999,POK),'BART JOB',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM SYSAFF=SC63
//*
//* Sample job to run a Java application in batch
//*
//INIT SET JAVA='java '                                         1
//         SET  CLASS='com.ibm.itso.sg246435.sqlj.Hello'        2
//         SET  ARGS='jdbc:db2:DB7Y bart b1ngo 30000 50000'     3
//         SET  STDOUT='/tmp/java.stdout.'
//         SET  STDERR='/tmp/java.stderr.'
//*
//JAVA     EXEC PGM=BPXBATCH,REGION=0K,PARM='SH &JAVA &CLASS &ARGS'    4
//STDOUT   DD   PATH='&STDOUT.&SYSUID',PATHOPTS=(OWRONLY,OCREAT,OTRUNC)5
//STDERR   DD   PATH='&STDERR.&SYSUID',PATHOPTS=(OWRONLY,OCREAT,OTRUNC)
/*
//* Copy stderr and stdout to JES data set
//CPYOUT   EXEC PGM=IKJEFT01,COND=EVEN                          6
//HFSOUT   DD   PATH='&STDOUT.&SYSUID',PATHDISP=(DELETE,KEEP)
//HFSERR   DD   PATH='&STDERR.&SYSUID',PATHDISP=(DELETE,KEEP)
//STDOUT   DD   SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//STDERR   DD   SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//SYSTSPRT DD   DUMMY
//SYSTSIN  DD   *
```

```
 ocopy indd(HFSOUT) outdd(STDOUT)
 ocopy indd(HFSERR) outdd(STDERR)
/*
//
```

Notes on Example 8-2 on page 147:

1. Invocation of the JVM executable program.
2. The main class of the Java program.
3. Command-line arguments to the program.
4. Invokes BPXBATCH to run the JVM as a shell script. When running as a shell script, at initialization time the .profile of the user ID running the program is invoked. Make sure the proper setup is done. See “Setting up the JDBC/SQLJ environment variables” on page 70 for details on setting up your .profile for JDBC/SQLJ usage.
5. Creates HFS files for standard output and standard error.
6. Invoke TSO and run the ocopy program to copy the standard output and standard error HFS files to a SYSOUT dataset. When the step completes successfully, the HFS files will be deleted.

Make sure your .profile is set up correctly (and includes the place where the Hello program is located in the CLASSPATH).

Note that JCL restricts the length of the PARM positional parameter to 100 characters so the combined JVM executable name, main class name, and arguments cannot exceed 95 characters (100 minus length of ’SH’ and three blanks).

9

The SQLJ program preparation process

In the previous chapter we wrote our first SQLJ application and executed it. We limited the discussion to the steps you need to perform, without getting into the details of why these steps are required, and what the SQLJ statements really mean.

This chapter explains why we had to perform the steps we did. It provides a discussion of the SQLJ program preparation process; that is, the steps that have to be performed in order to precompile an SQLJ file, to customize the generated profiles, and to bind the generated packages against the database.

The following chapter (Chapter 10, “SQLJ tutorial and reference” on page 175) deals with what all the SQLJ language constructs mean.

9.1 Program preparation in other languages

In order to better understand the program preparation process for SQLJ, we first discuss how the program preparation process works for other languages with embedded SQL support, such as COBOL. This process is illustrated in Figure 9-1 on page 151. The straight lines indicate steps that occur at compile time; the dashed line indicates runtime.

1. The original source file is translated by the DB2 precompiler into a compilable COBOL file in which the embedded SQL statements (from the original source program) have been replaced by calls to the DB2 language interface. The precompiler generates a *consistency token* into the COBOL file, which is later used at runtime to verify that the information in the DB2 catalog is still in sync with the program.

   The precompiler also generates a DataBase Request Module (DBRM), which is basically a file containing all SQL statements in the program. (The DBRM contains the same consistency token.)

2. The precompiled COBOL program is then translated into an object module (not explicitly shown in the figure), and the object module is then linked to form a load module.
3. The DBRM produced in step 1 is bound against the database. The BIND utility checks for correct SQL statement syntax, that all required objects (tables etc.) exist and that the person performing the bind step has sufficient permissions to execute the statements in the DBRM.

   DB2 also determines an *access path* to execute the SQL statements in the DBRM. An access path is the method selected by the DB2 optimizer for accessing data from a table. When binding the DBRM into a package, you also specify the isolation level to be used for that package (UR, CS, RS, or RR).

4. The result of the bind step is known as a package, which is an object in the database (it is stored in the DB2 catalog and directory) containing all the information from the DBRM file, as well as the access path. The information in the package includes the consistency token produced in step 1.
5. At runtime, DB2 uses the information in the package in order to execute the SQL statements in the program. It compares the consistency token in the program with the one in the DB2 catalog; in case of a mismatch, an error is reported, because this indicates that the program had been retranslated (precompiled again), but the DBRM had not been bound into a package.

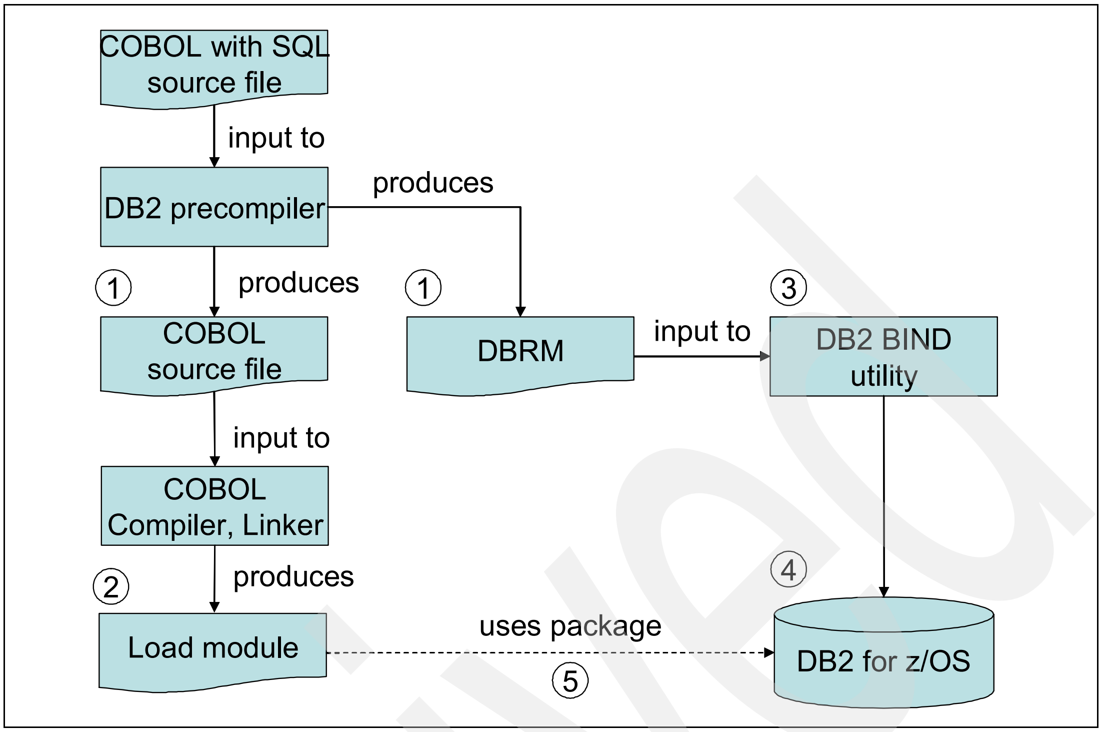

Figure 9-1 Program preparation in COBOL

The important thing to understand about packages and DBRMs is that packages are needed at runtime, whereas DBRMs are not. If the bind step is not performed after precompilation, the program will fail at runtime, because the package that is needed to execute the SQL does not exist. The DBRM, on the other hand, could be deleted after the bind step (although this is usually not done in practice).

9.2 Overview of the SQLJ program preparation process

The SQLJ program preparation process is similar to the program preparation process for (embedded) static SQL programs in other languages, like COBOL (see "Program preparation in other languages" on page 150). The process is illustrated in Figure 9-2 on page 152, where again, the straight lines indicate compile time and the dashed lines indicate runtime. The following steps are performed:

1. The SQLJ translator (also called SQLJ preprocessor) translates the original SQLJ source file into a Java file. SQL statements in the source file are replaced by calls into the SQLJ runtime library. The translator also produces one or more serialized profiles (.ser files). To be precise, it creates a separate serialized profile for each context class used in the program; see "Using more than one context class" on page 194 for details.
2. The Java compiler translates the preprocessed source file into class files. At this point, the program can be run successfully, but it will use dynamic SQL at runtime.
3. Optionally, the serialized profiles can (and should) be customized by the profile customizer. The profile customizer updates the .ser files and, by default, also creates a package in the DB2 database (catalog). This process enables the program to use static SQL at runtime.

4. When the program is run, the SQLJ runtime reads the serialized profile and uses the information in the profile to execute the SQL statements in the program. When the profiles have been customized, true static SQL is used; otherwise, the runtime uses dynamic SQL.

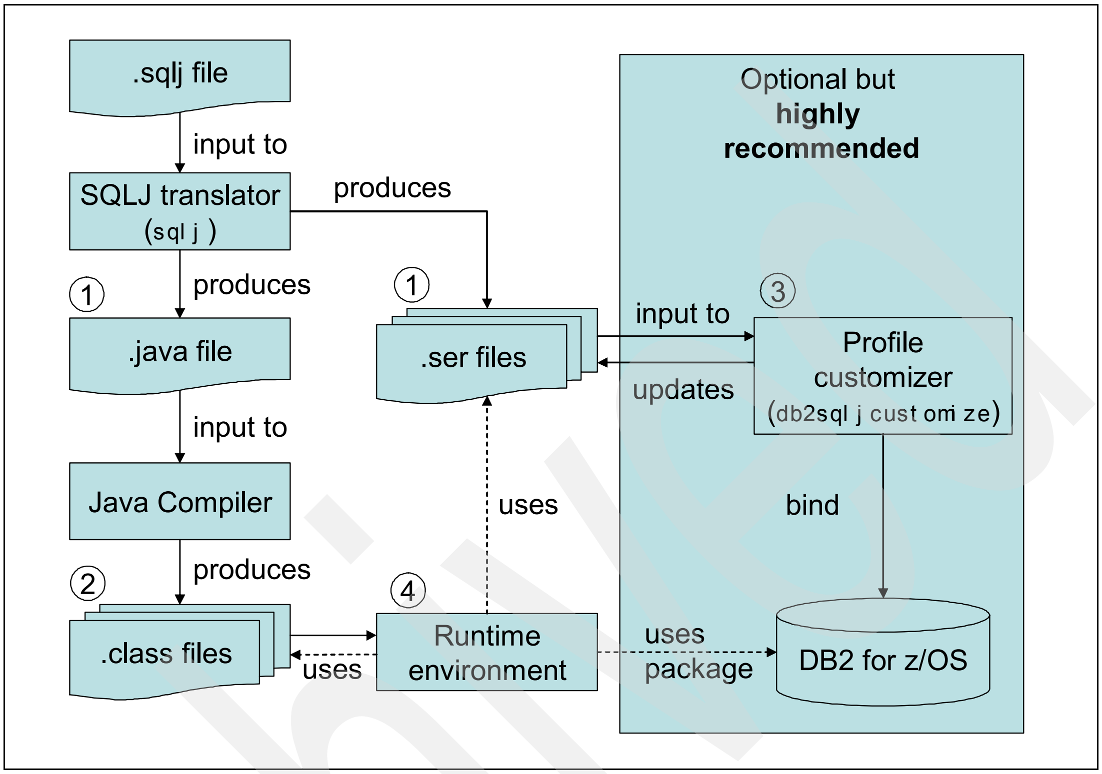

Figure 9-2 The SQLJ program preparation process - Overview

Although there are a lot of similarities, if you compare the SQLJ program preparation process, with the process for other languages, you will note some important differences:

- First, unlike DBRMs, the serialized profiles are needed at runtime (you will receive an `SQLException` if the runtime cannot find them).
- Second, the customization and bind steps are optional (but recommended), whereas in traditional programming languages they are required.

In the following sections, we explain these differences in more detail.

9.2.1 The SQLJ translator

The first step in SQLJ program preparation is to invoke the SQLJ translator, which, by the way, is a pure Java program. The translator takes an SQLJ source file, checks for correct Java and SQLJ syntax, and produces a compilable Java source file, and one or more SQLJ profiles.

In WSAD 5.1, the SQLJ translator is automatically invoked whenever you save an SQLJ file or (re)build the project, provided the enclosing project had been enabled for SQLJ support, and you have selected to **Perform build automatically on resource modification** in the Workbench preferences.

> **Note:** Only when your SQLJ source file does not contain any SQL statements, no profile will be produced.

During preprocessing, the translator invokes a Java parser and an SQL parser to check Java and SQLJ constructs for syntactical correctness. However, the translator does not catch all possible errors in the source file. For example, it does not detect missing variable declarations, except if these variables are used as host variables or in host expressions. It also does not recognize syntax errors in the SQL statements proper; this type of error will be caught by the profile customizer and/or binder.

The generated Java file looks pretty much the same as the source SQLJ files, except:

- SQLJ iterator and context declarations are converted into class declarations.
- Embedded SQL statements are converted into calls to the SQLJ runtime library. (They appear as comments in the translated Java file.)

The output of the SQLJ translator is platform independent; that is, you do not need to re-translate your SQLJ source files when targeting a database from a different vendor. This is true for both the generated Java source file and the profiles.

9.2.2 More about profiles

The generated profiles contain information about all the SQL statements in the program. A profile consists of one or more profile entries, each of which describes one SQL statement. Each profile entry in turn contains the type of statement, the number and type of parameters and results, the actual SQL text, and zero or more profile customizations (explained below). In this respect, a serialized profile is much like a DBRM.

A profile is actually an instance of the SQLJ runtime class `sqlj.runtime.profile.Profile`, and the .ser file produced by the SQLJ translator is the serialized form of that object, produced by the standard Java serialization mechanism.

Therefore, you can reconstitute the profile object by deserializing it:

```
import sqlj.runtime.profile.Profile;
...
ObjectInputStream ois = new ObjectInputStream(new FileInputStream(serfile));
Profile p = ois.readObject();
```

This is exactly what the SQLJ runtime library does when the program is executed.

> **Tip:** A convenient way to try this from within WSAD is to use a scrapbook page, as follows:
> 1. Open a scrapbook page (select **File** → **New** → **Scrapbook page**).
> 2. Type the code above (replace `serfile` with the name of a profile in your file system). The class names must be fully qualified; the easiest way to do this is to hit Ctrl+Space after the class name.
> 3. Select all (press Ctrl+A) then select **Run** → **Inspect**.
>
> An Inspector window opens and allows you to look at the profile's contents.

Another way to look at the contents of a serialized profile is to use the **`db2sqljprint`** utility. For more information see "The DB2 profile printer" on page 156.

As mentioned before, serialized profiles are platform independent; in other words, their format and contents are standardized (precisely because they are created by converting Java objects into their serialized form).

On the other hand, there must be a way to place vendor-specific information into the profiles. Otherwise, it would be impossible to support vendor-specific extensions. This is where profile customizations enter.

9.3 The DB2 profile customizer

The profile customizer (*db2sqljcustomize*) adds customization information to each entry in the profile. As mentioned before, this customization information is DBMS specific, and additional program preparation steps may be necessary before the customized program can successfully be run against the database.

In the case of DB2, during profile customization, a DB2 package is created in the database catalog (actually, four packages by default, one for each isolation level). The serialized profile is also updated by the customization process. The DB2 customization information includes the package name (supplied when invoking the profile customizer command); a timestamp indicating when the customization took place; and a *consistency token,* used to ensure that the database package matches the profile.

When an SQLJ implementation reads a serialized profile at runtime, it checks for available customizations that are compatible with that particular implementation. When no such customization is found, the SQLJ runtime falls back to using dynamic SQL; that is, it reads the actual SQL text of the statements, and prepares and executes these statements dynamically using JDBC calls.

This is in fact one of the beauties of SQLJ, as opposed to other languages with embedded SQL support. Whereas in COBOL, for example, you are required to bind your program against the database after translation—otherwise, you get a runtime error when executing it—this is not the case with SQLJ. Rather, you simply translate the program and the SQLJ runtime will figure out whether or not the program has been customized. This is very convenient during program development, since you can omit the customization (and bind) step until your program is ready for deployment. (This is in fact what we did with the Hello sample in the previous chapter. When we executed it, it ran as dynamic SQL because we had not customized it (yet).)

Important: However, if the profile has been customized for DB2 (customization information present in the serialized profile), and for some reason the package has not been bound against the database during (or after) customization, you receive a -805 SQL error (package not found) at runtime.

You will also receive this error when the package had been bound before, but the SQLJ file was retranslated and re-customized without rebinding the package. In this case, the consistency token recorded in the profile does not match the one in the database.

So once the profile has been customized for DB2, you must have a matching package bound in the database. In this case, SQLJ runtime will not revert to using JDBC.

The profiles remain platform independent, even after customization, because SQLJ implementations are required to ignore customization information they do not know about. In fact, it is possible to customize one profile several times, once for each target database system. This makes it possible to run the program unchanged against, say, DB2 and Oracle databases. However, all DB2 implementations across the DB2 family use the same customization. You do not have to customize again when switching from a DB2 for Linux, Unix, and Windows to DB2 for z/OS and OS/390 when using the DB2 Universal Driver.

9.3.1 Isolation levels

SQLJ allows you to change the isolation level at the beginning of a new transaction, using the `SET TRANSACTION ISOLATION LEVEL` clause (see “Executable statements” on page 176). However, each package in DB2 is created to use a specific isolation level. Therefore, the profile customizer, by default, creates four packages, one for each isolation level. The names of these packages consist of the root package name, which you specify when invoking the profile customizer, and a suffix of 1 through 4, corresponding to isolation levels UR, CS, RS and RR, respectively. Since a package name in DB2 can be up to eight characters long, the root package name must not exceed seven characters. For example, if your root package name is HELLO, the profile customizer creates the following packages:

`HELLO1`	For isolation level UR (uncommitted read)
`HELLO2`	For isolation level CS (cursor stability)
`HELLO3`	For isolation level RS (read stability)
`HELLO4`	For isolation level RR (repeatable read)

If you are sure that your program will always use one specific isolation level, you can free the packages created for the other isolation levels. Alternatively, you can tell the profile customizer to only create one package, for a specific isolation level. Of course, if you switched to a different transaction isolation level at runtime, you will receive a -805 error (Package not found). You do so by specifying the following option on the **`bind`** command:

```
-singlepkgname pkgname -bindoptions "isolation isolation level"
```

Where:

- *pkgname* is the name of the package.
- *isolation level* it the isolation level you want the package to use, for example, CS. Note that we use ‘DB2 notation’ , UR, CS, RS, RR, not ‘JDBC notation’ 1, 2, 3, 4, to specify the isolation level.

> **Note:** You can also specify the -singlepkgname and -bindoptions parameters in WSAD V5.1 when using the SQLJ tooling, the same way as you specify the -collection option (see Figure 9-3 on page 159).

9.3.2 Why online checking is good for you

By default, the profile customizer runs with online checking enabled, if you entered a value in the URL field in the SQLJ Customization Script dialog (Figure 9-3 on page 159).

When online checking is enabled, the SQLJ customizer queries the DB2 catalog on the target database server in order to verify whether your SQL statements will be able to run on that server.

Online checking will recognize errors or potential problems such as:

- Misspelled names (including table or view names, column names, function names, etc.)
- Lack of authorization (for example, an INSERT statement on a table for which you (the package owner) do not have INSERT authority)
- Data type mismatches (for example, trying to UPDATE a numeric column with the value of a String host variable)
- Specifying a host variable of Java primitive type to receive the value of a nullable column

Without online checking, all of these errors will go unnoticed until bind time or even runtime.

Also, enabling online checking is strongly recommended for performance reasons (see the detailed discussion in “Always customize with online checking enabled” on page 236). If online checking is not enabled, the DB2 optimizer may choose a poor access path due to lack of information, resulting in badly performing queries.

9.4 The DB2 profile binder

As explained in “The DB2 profile customizer” on page 154, the DB2 profile customizer by default also binds the profile against the database, producing (or updating) a database package for that profile.

If an existing application needs to be deployed against another database (of the same ‘family’) there is no need to recustomize the profile again. You can just take the existing customized profile and bind it against the new database (or DB2 for z/OS subsystem).

To rebind an existing, customized profile, you can use the **db2sqljbind** tool.

> **Note:** Unfortunately, at the time of writing, rebinding without recustomizing is not supported by the SQLJ tooling in WSAD V5.1.

9.5 The DB2 profile printer

The DB2 profile printer (**db2sqljprint**) is a tool you can use to print the information contained in a serialized profile in human-readable form. Essentially, what the profile printer does is to read a serialized profile, de-serialize it, and print all profile entries, including DB2 customization information if present.

Example 9-1 shows part of the **db2sqljprint** output for the customized profile of our simple Hello program.

Example 9-1 Output of db2sqljprint

```
===================================================
printing contents of profile com.ibm.itso.sg246435.sqlj.Hello_SJProfile0
created 1059525326023 (2003-07-29)
associated context is sqlj.runtime.ref.DefaultContext
profile loader is sqlj.runtime.profile.DefaultLoader@32ae0b42
contains 1 customizations                                                  1
com.ibm.db2.jcc.sqlj.Customization@2eaa4b42
original source file: Hello.sqlj
contains 1 entries
===================================================
profile com.ibm.itso.sg246435.sqlj.Hello_SJProfile0 entry 0
#sql { SELECT LASTNAME                  , FIRSTNME                  , SALARY
FROM DSN8710.EMP                 WHERE SALARY BETWEEN  ?  AND  ?                ORDER
BY LASTNAME, FIRSTNME             };
line number: 45
PREPARED_STATEMENT executed via EXECUTE_QUERY
role is QUERY
descriptor is null
contains 2 parameters
1. mode: IN, java type: java.math.BigDecimal (java.math.BigDecimal),
   sql type: NUMERIC, name: min, marker index: 152
2. mode: IN, java type: java.math.BigDecimal (java.math.BigDecimal),
   sql type: NUMERIC, name: max, marker index: 160
result set Type is POSITIONED_RESULT
```

```
result set name is com.ibm.itso.sg246435.sqlj.Hello$EmployeeIterator
contains 3 result columns
1. mode: OUT, java type: java.lang.String (java.lang.String),
   sql type: VARCHAR, name: getCol1, marker index: -1
2. mode: OUT, java type: java.lang.String (java.lang.String),
   sql type: VARCHAR, name: getCol2, marker index: -1
3. mode: OUT, java type: java.math.BigDecimal (java.math.BigDecimal),
   sql type: NUMERIC, name: getCol3, marker index: -1

--------- EntryInfo custom data ----------
(... omitted ...)
===================================================

--------- DB2 ProfileData custom data ----------                         2
Package Name: HELLO
Package Version: null
Package Collection: SG246435
Is Default Collection: false
Package Consistency Token, hex format: 4341616a52434770
Package Consistency Token, character format: CAajRCGp
```

Notes on Example 9-1 on page 156:

1. This profile has been customized.
2. DB2-specific customization information, such as the collection, package name and consistency token.

9.6 Preparing an application to use static SQL

Until now, we ran our Hello application with uncustomized serialized profiles, meaning that the DB2 calls were executed as dynamic SQL. As explained in “More about profiles” on page 153, this is very convenient during program development since you do not have to care about binding your program against the database.

In this section we customize the serialized profile generated for the program and verify that the program now uses static SQL to access the database.

We first show how to do this using WSAD, but also how to do it ‘the hard way’ using manual commands.

9.6.1 Preparing SQLJ programs to use static SQL through WSAD

Unlike preprocessing (SQLJ translation), which happens immediately whenever you save a changed SQLJ source file in the WSAD editor, the customization process is triggered from an *Ant script*, which you have to invoke explicitly. The good news is that WSAD generates this script for you.

A brief introduction to Ant

As the Ant user manual puts it, “Apache Ant is a Java-based build tool. In theory, it is kind of like *make*, without *make's* wrinkles”. Why is it called Ant? Because it is a little thing that can build big things.

If you are familiar with make, you probably know how difficult it can be to write a makefile that works on each system you are targeting. Also, make is purely timestamp based—it compares the timestamps of targets against the timestamps of dependents to decide whether a target has to be rebuilt.

Ant, on the other hand, is platform independent and much more flexible and extensible. It is centered around the concept of *tasks*, which are Java classes that perform a certain job. Ant comes with a number of useful predefined tasks, for example, to compile Java source code, to do FTP transfers, to perform unit tests, and many more.

Tasks are executed under the control of *targets*. To describe what tasks must be performed in order to build a given target, you write a *buildfile* in XML. Each buildfile must specify at least one target; of course, you can have many targets, and those targets may have *dependencies*. For example, in order to export Java class files to another system, you first have to compile the corresponding Java sources files so the export target will be dependent on the compile target.

WSAD comes with built-in Ant integration, so starting Ant from within WSAD is as easy as pushing a button. This eliminates the necessity to leave WSAD, go to the DB2 command window, run the customizer, go back to WSAD, and refresh the workspace.

Customization script generation

Let us now try to get the Hello.sqlj program to run using static SQL.

Right-click the project then select **Properties**. Select the **SQLJ Customization Script** panel, and fill out the URL field (Figure 9-3 on page 159).

Important: Filling out the URL field is important. When you leave it blank, customization is likely to fail. The JCC driver, by default, tries to bind the packages against the database during customization (-automaticbind YES is the default). In addition, the driver will do online checking (-onlinecheck YES is the default). Online checking is very important. See “Why online checking is good for you” on page 155 for more details.

The dialog also allows you to fill out the User and Password fields. However, as this is a possible security exposure, we suggest that you leave them blank.

We also recommend (and assume in the following sections) that you enter the following in the Options field:

```
-collection SG246435
```

This causes the packages to be created in a collection named SG246435 (if you do not specify this option, they are created in the NULLID collection). Your shop probably has conventions for collection names, so you may want to replace SG246435 with another name that adheres to these conventions. The -collection flag also causes the named collection to be registered in the serialized profile. When using the T4 driver, this information is passed to the server and is used to search for the correct package at the server.

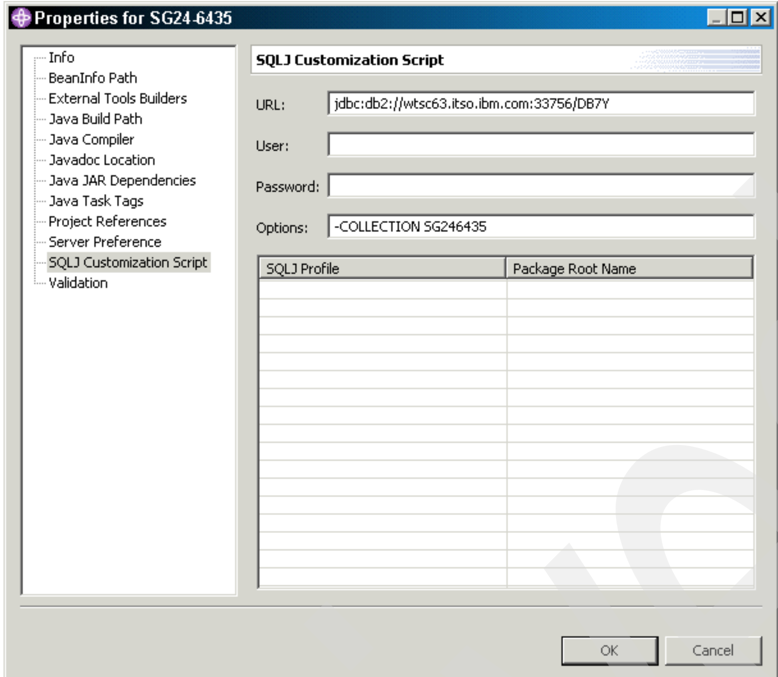

Figure 9-3 SQLJ customization script properties

Attention: Make sure to specify the collection name in uppercase. Otherwise you get the error message shown in Example 9-2 when executing the Ant script.

Example 9-2 Customization error using lowercase characters

```
Buildfile: C:\wsad51\workspace\SG24-6435\SQLJAntScripts\sqlj.customize.xml

com.ibm.itso.sg246435.sqlj.Hello:
        [java] [ibm][db2][jcc][sqlj]
        [java] [ibm][db2][jcc][sqlj] Begin Customization
        [java] [ibm][db2][jcc][sqlj] Loading profile: com\ibm\itso\sg246435\sqlj\Hello_SJProfile0
        [java] [ibm][db2][jcc][sqlj] Customization complete for profile
                                     ..\\com\ibm\itso\sg246435\sqlj\Hello_SJProfile0.ser
        [java] [ibm][db2][jcc][sqlj] Begin Bind
        [java] [ibm][db2][jcc][sqlj] Loading profile: com\ibm\itso\sg246435\sqlj\Hello_SJProfile0
        [java] [ibm][db2][jcc][sqlj] Binding package HELLO1 at isolation level UR
        [java] [ibm][db2][jcc][sqlj] The DDM parameter value is not supported.  DDM parameter code point
                                     having unsupported value : 0x2112
        [java] [ibm][db2][jcc][sqlj] ***Bind process has failed!***
        [java] BUILD FAILED: file:C:/wsad51/workspace/SG24-6435/SQLJAntScripts/sqlj.customize.xml:15: Java
                             returned: -1
Total time: 5 seconds
```

This error occurs when binding against a DB2 V6 or V7 subsystem. The DRDA level that they support requires the collection ID to be in upper case. This is no longer necessary when binding against a V8 system. A future version (most likely FixPak 5) of the JCC driver will produce an enhanced error message that makes it easier to identify the cause of the problem.

Click **OK**. You can select the properties that we just specified by expanding the (newly created) SQLJAntScripts directory and double-clicking the **sqlj.project.properties** file. It should look somewhat like Figure 9-4.

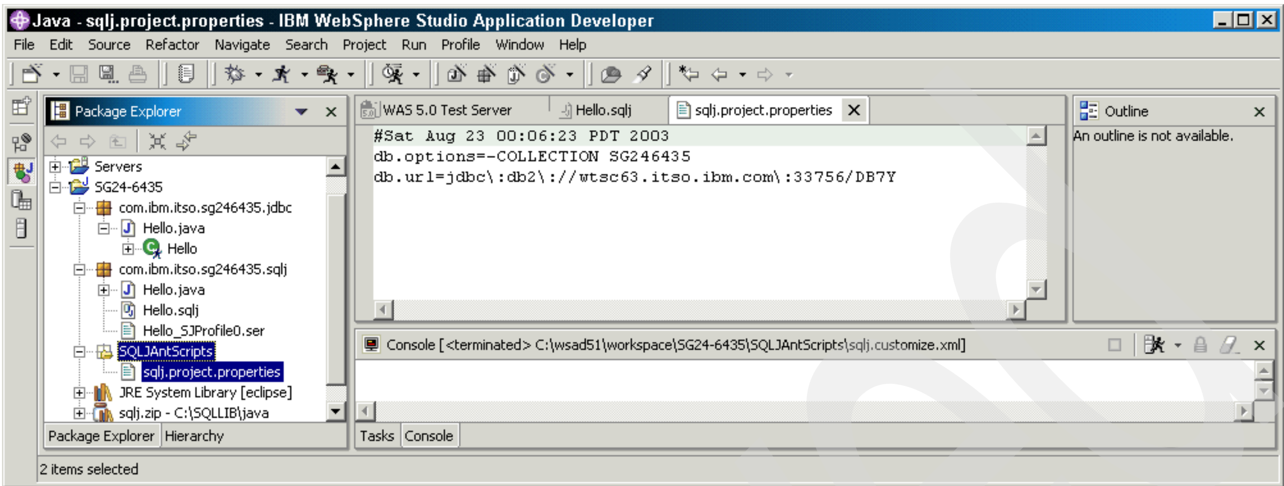

Figure 9-4 Sqlj.project.properties

> **Important:** If you specified values for the User and Password fields in the previous step, these values would have been stored in the properties file in plain text. Obviously, storing passwords in plain text is not a good idea, especially when you are sharing the properties file in a CVS repository.

Next, right-click the project again and select **Generate SQLJ Customization Script**. WSAD now generates the build script, a file called sqlj.customize.xml. It also modifies the properties file generated in the previous step, adding one property for each SQLJ source file that stores the root package name for that SQLJ source file.

Now open the **Project Properties** dialog again (right-click the project and select **Properties**). The list box in the lower half of the panel now shows all the SQLJ profiles in the project, and the corresponding package root name that has been assigned to that profile. Since the default root name Hello_0 is somewhat arbitrary, you should change it to HELLO (see Figure 9-5 on page 161). The changes will be reflected in the sqlj.project.properties file (which you also could have changed manually, instead of using the properties dialog). Then click **OK**.

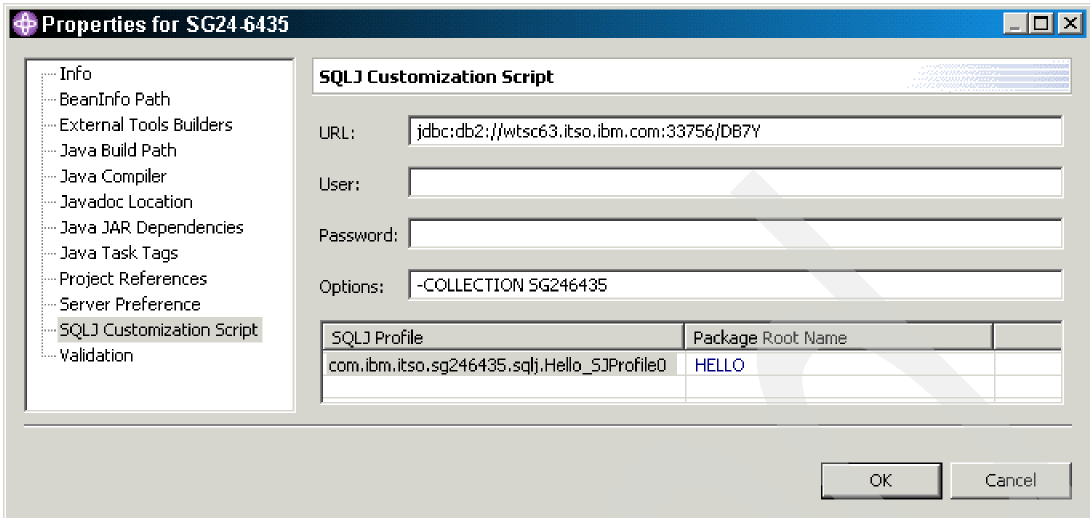

Figure 9-5 Changing the root package names

Running the customization script

Now we are ready to run the customization script. Right-click the script (**sqlj.customize.xml**) and select **Run Ant...**. In the dialog that opens, switch to the Properties pane. Here we configure the database username and password. Using the **Add...** button, add two properties called "db.user" and "db.password" and their respective values (Figure 9-6).

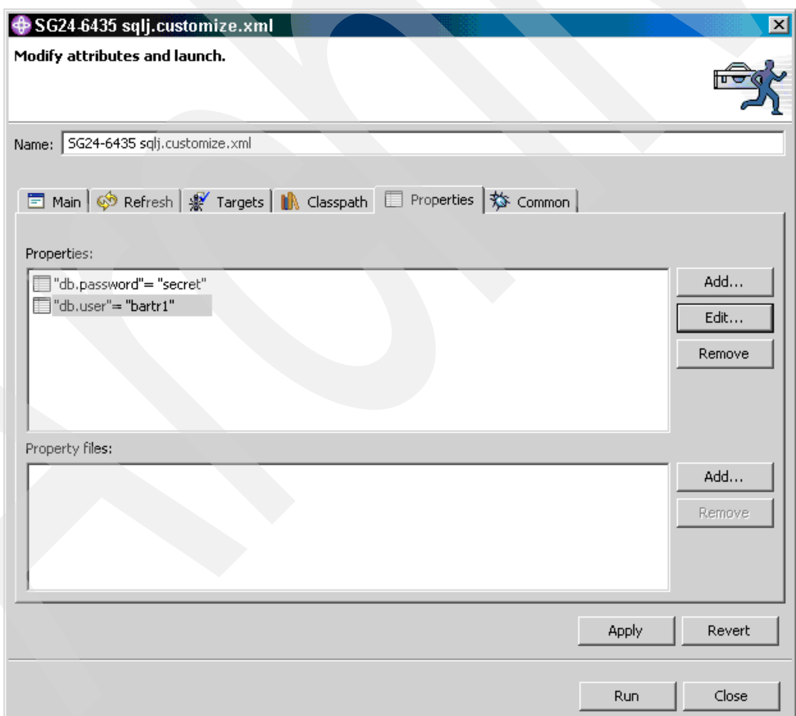

Figure 9-6 Setting the user name and password properties

Now click the **Run** button at the bottom. WSAD executes the build script and displays the messages in the Console view (Example 9-3 on page 162).

Example 9-3 Output from running the customization script

```
Buildfile: C:\wsad51\workspace\SG24-6435\SQLJAntScripts\sqlj.customize.xml

com.ibm.itso.sg246435.sqlj.Hello:
        [java] [ibm][db2][jcc][sqlj]
        [java] [ibm][db2][jcc][sqlj] Begin Customization
        [java] [ibm][db2][jcc][sqlj] Loading profile:
                                       com\ibm\itso\sg246435\sqlj\Hello_SJProfile0
        [java] [ibm][db2][jcc][sqlj] Customization complete for profile
                                       ..\\com\ibm\itso\sg246435\sqlj\Hello_SJProfile0.ser
        [java] [ibm][db2][jcc][sqlj] Begin Bind
        [java] [ibm][db2][jcc][sqlj] Loading profile:
                                       com\ibm\itso\sg246435\sqlj\Hello_SJProfile0
        [java] [ibm][db2][jcc][sqlj] Binding package HELLO1 at isolation level UR
        [java] [ibm][db2][jcc][sqlj] Binding package HELLO2 at isolation level CS
        [java] [ibm][db2][jcc][sqlj] Binding package HELLO3 at isolation level RS
        [java] [ibm][db2][jcc][sqlj] Binding package HELLO4 at isolation level RR
        [java] [ibm][db2][jcc][sqlj] Bind complete for
                                       com\ibm\itso\sg246435\sqlj\Hello_SJProfile0

customizeAll:
BUILD SUCCESSFUL
Total time: 6 seconds
```

Tip: The Ant script uses the standard Windows CLASSPATH to search for the SQLJ customizer. Therefore, make sure that db2jcc.jar and db2jcc_license_cisuz.jar appear in the standard windows CLASSPATH.

Verifying that the packages have been created

Next we verify that the SQLJ profile customizer indeed created the packages in the database for our two SQLJ source files. Start your favorite interactive query tool (for example, QMF or the DB2 Command Center), and issue the following query:

```
SELECT COLLID, NAME, CREATOR, TIMESTAMP
  FROM SYSIBM.SYSPACKAGE WHERE CREATOR = 'YOUR_USERID'
```

Assuming that you never created any other packages in the database, your output will look similar to Example 9-4.

Example 9-4 Excerpt from SYSIBM.SYSPACKAGE catalog table after profile customization

```
COLLID            NAME      CREATOR   TIMESTAMP
---------+---------+---------+---------+---------+---------+--------
SG246435          HELLO1    BART      2003-08-08-19.35.20.341008
SG246435          HELLO2    BART      2003-08-08-19.35.20.481462
SG246435          HELLO3    BART      2003-08-08-19.35.20.613669
SG246435          HELLO4    BART      2003-08-08-19.35.20.745443
DSNE610I NUMBER OF ROWS DISPLAYED IS 4
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
```

Verifying that your application uses static SQL

The fact that a package exists is of course no guarantee that the package is actually being used. In order to verify that our SQLJ Hello application is really using static SQL, you can use your DB2 monitor. After the customization, run the program again from WSAD using the same run configuration that we set up in the previous chapter for the Hello SQLJ program.

Following are a number of screen captures from DB2 Performance Expert for z/OS to demonstrate that we are actually using static SQL. Figure 9-7 shows that the DISTSERV plan is used (as usual, since we come in through DRDA), and that we are executing the program HELLO2 (remember, 2 is the CS version of the package) from collection SG246435.

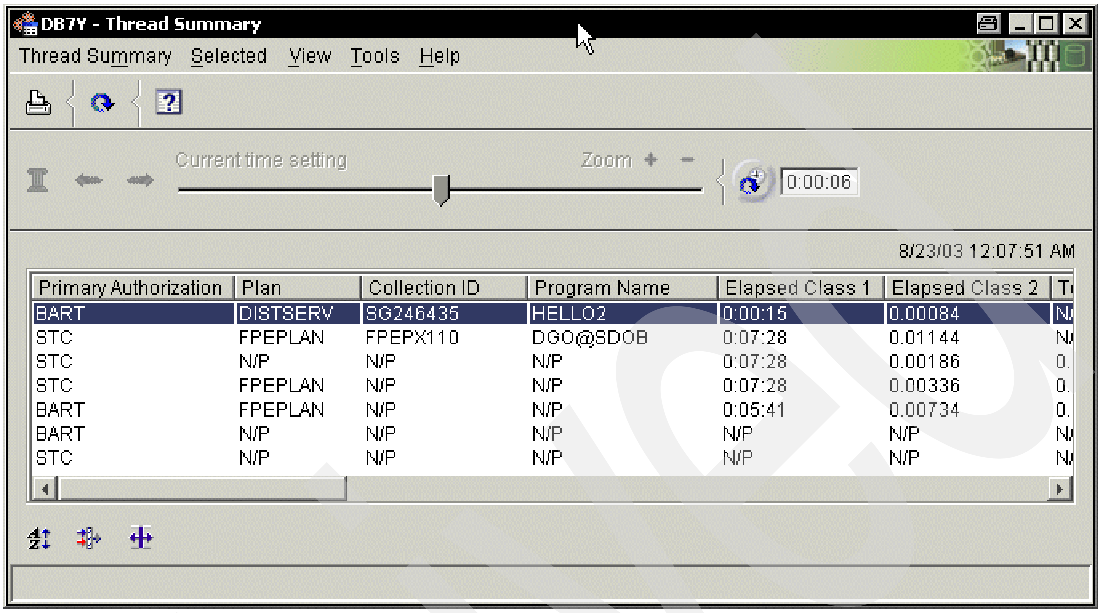

Figure 9-7 DB2 PE thread summary

In case you want to see more details, you can zoom into the thread, and look at the SQL statement that is being executed (Figure 9-8 on page 164). We see our cursor declaration from the SQLJ program. This is true static SQL indeed.

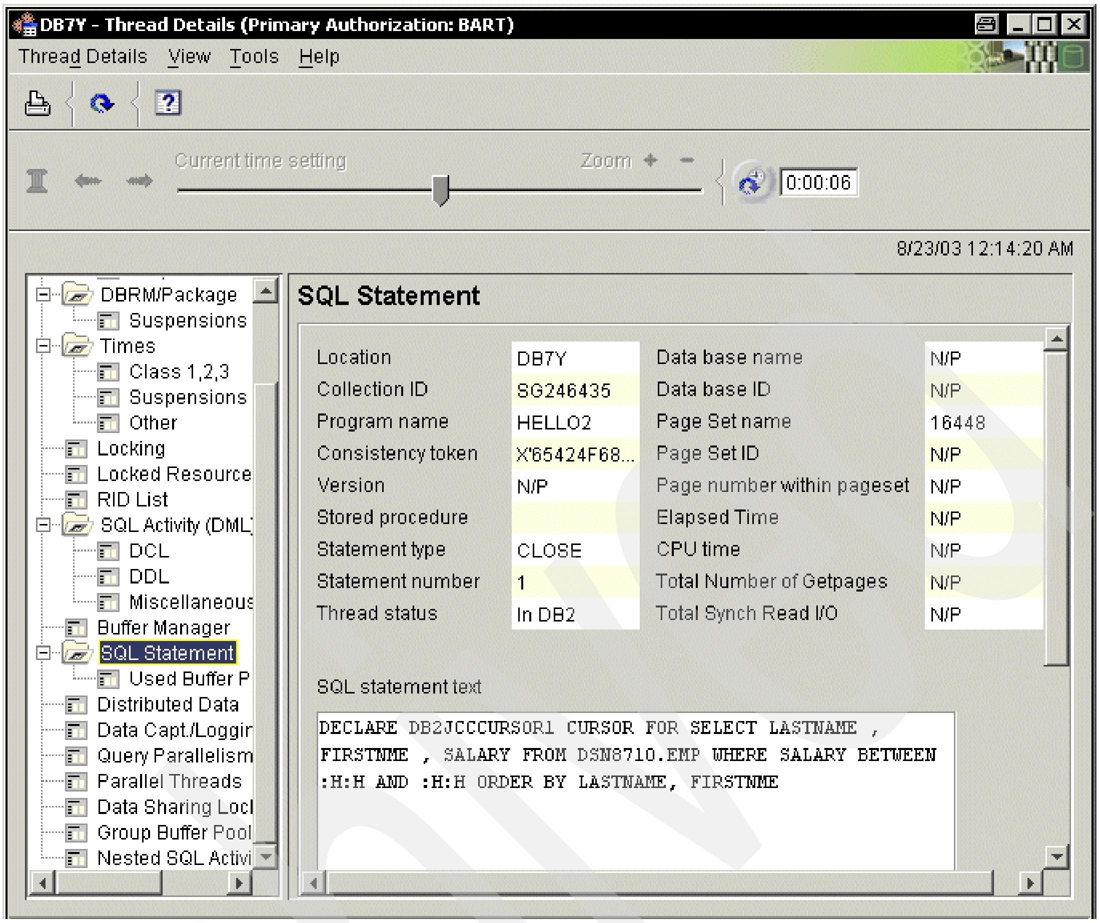

Figure 9-8 Currently execution SQL statement

If that is not proof enough, you can also look at the number and type of DML statements executed by the Hello program (Figure 9-9 on page 165). As you can see, no prepares were executed.

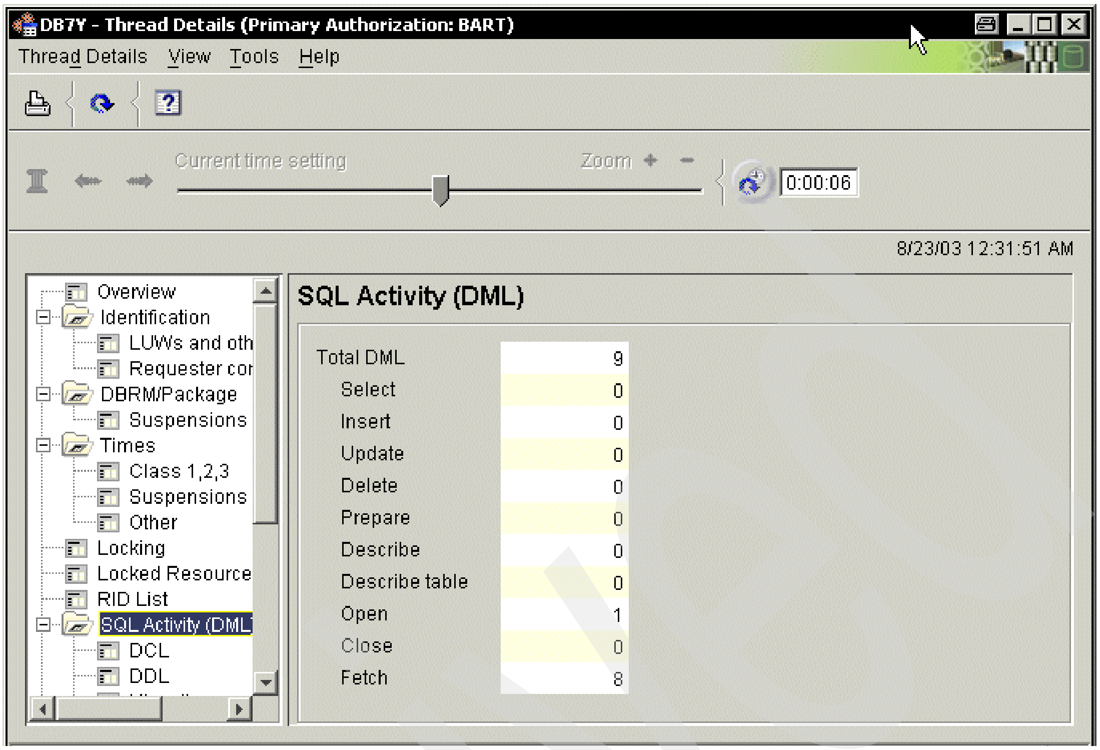

Figure 9-9 Type of DML statements executed by HELLO2

A more drastic way to demonstrate that the application now indeed uses static rather than dynamic SQL, is to free the DB2 packages.

In TSO, go to the DB2 Interactive panel, and select FREE PACKAGE (Option 5.7). Enter the collection ID and the package name, HELLO2, then press Enter (Figure 9-10).

> **Note:** Remember, the customizer creates four packages, one for each isolation level. Since the default isolation level after connecting to the database is CURSOR STABILITY (CS), it is sufficient for the sake of this example to only free the corresponding HELLO2 package that was bound with CS.

```
FREE PACKAGE                         SSID: DB7Y
COMMAND ===>

 1  Free ALL packages ......... ===>           (* to free authorized packages)

or
    Enter package name(s) to be freed:
 2  LOCATION NAME ............. ===>                   (Defaults to local)
 3  COLLECTION-ID ............. ===> SG246435            (Required)
 4  PACKAGE-ID ................ ===> HELLO2      (* to free all packages )
 5  VERSION-ID ................ ===> *
                                                (*, Blank, (), or version-id)
 6  ADDITIONAL PACKAGES? ...... ===> NO         (Yes to include more packages)

PRESS:  ENTER to process    END to save and exit     HELP for more information
```

Figure 9-10 Freeing the package

Then run EmployeeTest again. Not surprisingly, we receive the message:

```
com.ibm.db2.jcc.c.SQLException: DBRM OR PACKAGE NAME
DB7Y.SG246435.HELLO2.000000F29E5625A7 NOT FOUND IN PLAN DISTSERV. REASON 04.
```

This indicaties that the package referred to by the SQLJ profile could not be found, and thus verifies that the SQLJ runtime tried to run the program using static SQL.

To recreate the package, simply run the Ant buildfile again.

> **Note:** As shown in this example, once the profile has been customized for DB2, the package has to be present for the application to run. SQLJ only reverts to JDBC at runtime is no customization for that particular database that you are running against exists in the serialized profile.

9.6.2 Doing it yourself - Manual program preparation for static SQLJ

If you do not have a nice tool like WSAD that does all, or most, of the program preparation work for you, you can still do it by executing a set of commands, as we show hereafter.

In order to not have to retype our Hello.sqlj program, we export it into the file system from WSAD. Right-click the project and select **Export** → **File System** to get to the dialog shown in Figure 9-11. Select only the Hello.sqlj file, and make sure the “Create directory structure for files” option is checked. We export to a directory called TestSQLJ.

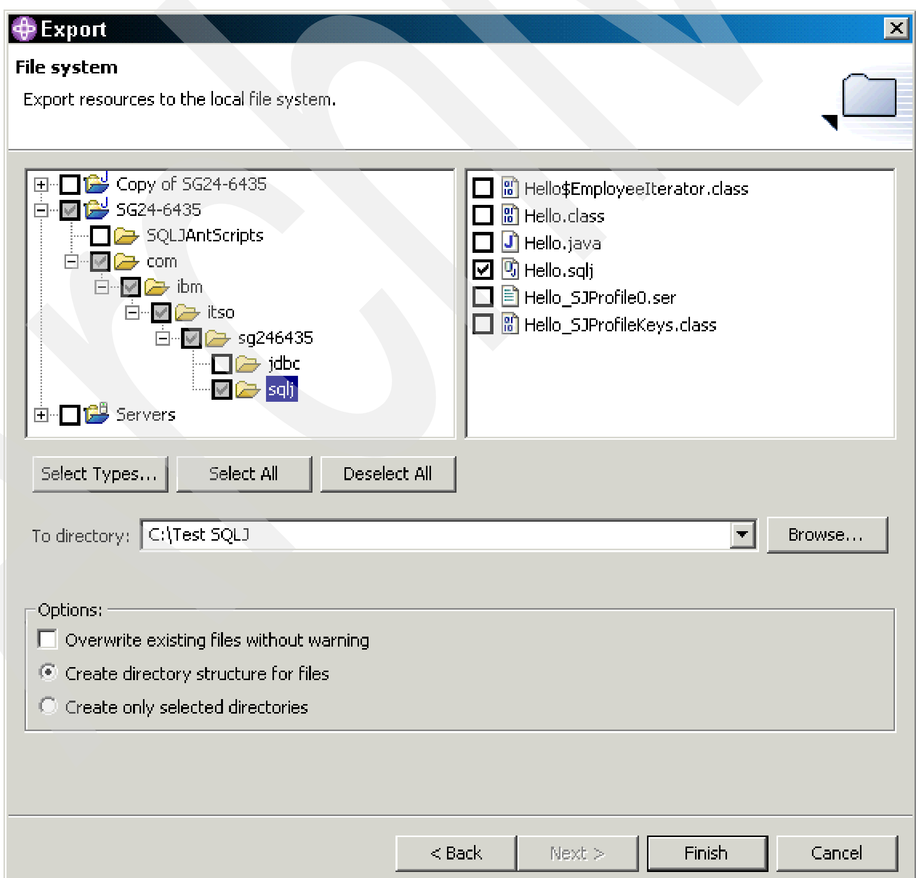

Figure 9-11 Export to the file system

The export creates the following directory structure (Figure 9-12), in sync with our Java package naming in the program.

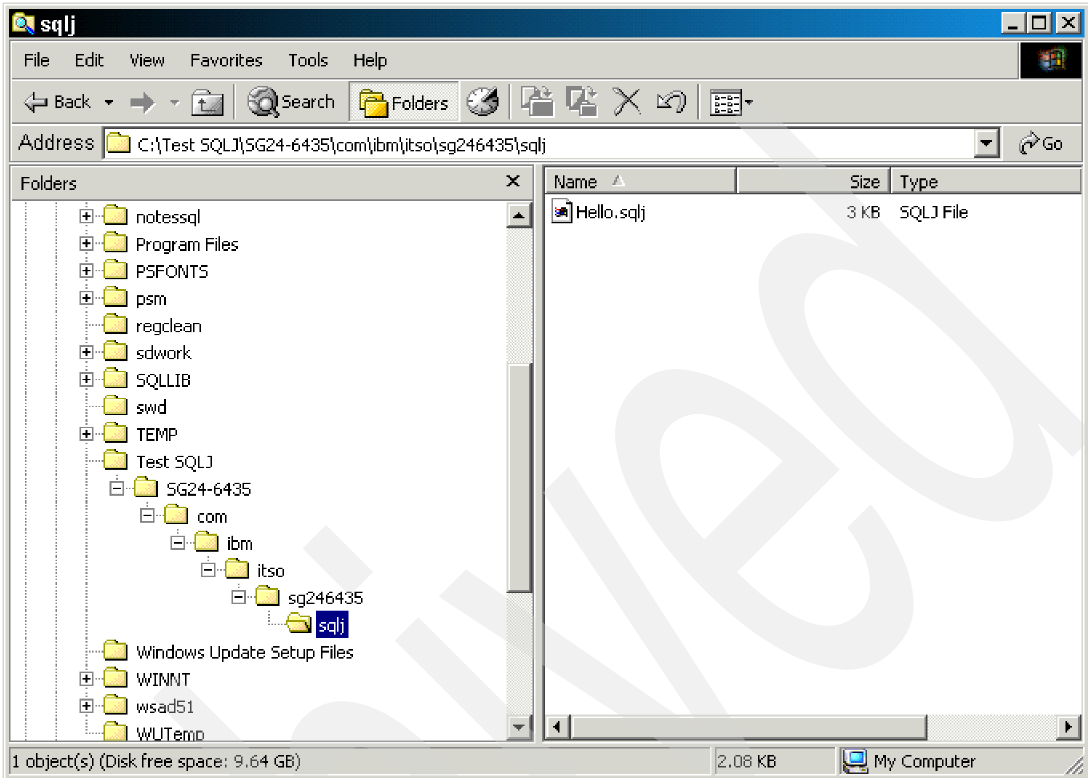

Figure 9-12 Directory structure after the export

Now we are ready to go. We show all individual steps that you have to do to get the program to run with static SQL.

Translate the program (sqlj)

The first step is to translate the SQLJ into a Java program using the SQLJ translator, using the following command:

```
C:\TestSQLJ\SG24-6435>sqlj -compile=false com/ibm/itso/sg246435/sqlj/Hello.sqlj
```

Make sure you are in the correct directory to do so. You could also have gone to the C:\TestSQLJ\SG24-6435\com\ibm\itso\sg246435\sqlj directory and issue:

```
C:\TestSQLJ\SG24-6435\com\ibm\itso\sg246435\sqlj>sqlj -compile=false Hello.sqlj
```

Notice that we use the -compile=false option. This prevents the Java compiler from being invoked by default. We only do this here to illustrate that the result of the sqlj translation are the following files:

```
Hello.java
Hello_SJProfile0.ser
```

Out of curiosity, let us have a look at the information that is stored in the serialized profile and use the db2sqljprint utility to print the contents of the Hello_SJProfile0.ser file (Example 9-5).

Example 9-5 Uncustomized serialized profile

```
===================================================
printing contents of profile com.ibm.itso.sg246435.sqlj.Hello_SJProfile0
```

```
created 1061630428017 (2003-08-23)
associated context is sqlj.runtime.ref.DefaultContext
profile loader is sqlj.runtime.profile.DefaultLoader@2909cbd6
contains 0 customizations
original source file: com/ibm/itso/sg246435/sqlj/Hello.sqlj
contains 1 entries
===================================================
profile com.ibm.itso.sg246435.sqlj.Hello_SJProfile0 entry 0
#sql { SELECT LASTNAME
          , FIRSTNME
          , SALARY
        FROM DSN8710.EMP
       WHERE SALARY BETWEEN  ?  AND  ?
       ORDER BY LASTNAME, FIRSTNME
      };
line number: 49
PREPARED_STATEMENT executed via EXECUTE_QUERY
role is QUERY
descriptor is null
contains 2 parameters
1. mode: IN, java type: java.math.BigDecimal (java.math.BigDecimal),
   sql type: NUMERIC, name: min, marker index: 90
2. mode: IN, java type: java.math.BigDecimal (java.math.BigDecimal),
   sql type: NUMERIC, name: max, marker index: 98
result set Type is POSITIONED_RESULT
result set name is com.ibm.itso.sg246435.sqlj.Hello$EmployeeIterator
contains 3 result columns
1. mode: OUT, java type: java.lang.String (java.lang.String),
   sql type: VARCHAR, name: getCol1, marker index: -1
2. mode: OUT, java type: java.lang.String (java.lang.String),
   sql type: VARCHAR, name: getCol2, marker index: -1
3. mode: OUT, java type: java.math.BigDecimal (java.math.BigDecimal),
   sql type: NUMERIC, name: getCol3, marker index: -1
===================================================
```

As you can see, the profile contains zero customizations.

Compile the program (javac)

As mentioned before, we only need to do this step because we used the -compile=false option on the sqlj command line.

```
C:\TestSQLJ\SG24-6435>javac com/ibm/itso/sg246435/sqlj/Hello.java
```

This creates the following files (in the com/ibm/itso/sg246435/sqlj subdirectory):

```
Hello.class
Hello$SJProfileKeys.class
Hello$EmployeeIterator.class
```

Note that the Java compiler also generates a class file for the iterator that we declared in the program (Hello$EmployeeIterator.class), and a class file for a utility class that is used internally by the SQLJ runtime (Hello$SJProfileKeys.class).

If we were to execute the program at this point, it would run using dynamic SQL. As our goal is to run with static SQL, we are not going to. However, in a development environment this is very useful to do validation of your SQL statements to make sure they return the correct data.

Customization (db2sqljcustomize)

This is the magic step that makes an sqlj program run statically against a DB2 system. We use the **db2sqljcustomize** utility for this purpose. The command is shown in Example 9-6.

Example 9-6 Customizing a serialized profile

```
C:\TestSQLJ\SG24-6435>db2sqljcustomize -url jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y -user
  bart -password b1ngo -rootpkgname HELLO -collection SG246435
  com/ibm/itso/sg246435/sqlj/Hello_SJProfile0.ser
[ibm][db2][jcc][sqlj]
[ibm][db2][jcc][sqlj] Begin Customization
[ibm][db2][jcc][sqlj] Loading profile: com/ibm/itso/sg246435/sqlj/Hello_SJProfil
e0
[ibm][db2][jcc][sqlj] Customization complete for profile com/ibm/itso/sg246435/s
qlj/Hello_SJProfile0.ser
[ibm][db2][jcc][sqlj] Begin Bind
[ibm][db2][jcc][sqlj] Loading profile: com/ibm/itso/sg246435/sqlj/Hello_SJProfil
e0
[ibm][db2][jcc][sqlj] Binding package HELLO1 at isolation level UR
[ibm][db2][jcc][sqlj] Binding package HELLO2 at isolation level CS
[ibm][db2][jcc][sqlj] Binding package HELLO3 at isolation level RS
[ibm][db2][jcc][sqlj] Binding package HELLO4 at isolation level RR
[ibm][db2][jcc][sqlj] Bind complete for com/ibm/itso/sg246435/sqlj/Hello_SJProfi
le0
```

Let us now look at some of the parms that we specified on the **db2sqljcustomize** command:

- `-url jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y` is the URL of the database we want to customize against. Note that we use the same notation as we use inside our program.
- `-user bart` is the user ID that is to be used to connect to the database.
- `-password b1ngo` is the password that is to be used to connect to the database.
- `-rootpkgname HELLO` is (the start of) the name that the static DB2 package will have when we bind it against the database.
- `-collection SG246435` is the name ofthe DB2 collection where that package will be bound into.
- `com/ibm/itso/sg246435/sqlj/Hello_SJProfile0.ser` is the name of the serialized profile that we want to customize.

Now that we have done a successful customization (and bind—invoking the binder is done by default) let us now look at the serialized profile again. Remember that customization not only binds a package against the database, but also updates the serialized profile. We use the **db2sqljprint** utility again. The result is shown in Example 9-7.

Example 9-7 Customized serialized profile

```
===================================================
printing contents of profile com.ibm.itso.sg246435.sqlj.Hello_SJProfile0
created 1061630428017 (2003-08-23)
associated context is sqlj.runtime.ref.DefaultContext
profile loader is sqlj.runtime.profile.DefaultLoader@2909cd92
contains 1 customizations                                                1
com.ibm.db2.jcc.sqlj.Customization@3de20d92                              2
original source file: com/ibm/itso/sg246435/sqlj/Hello.sqlj
contains 1 entries
===================================================
profile com.ibm.itso.sg246435.sqlj.Hello_SJProfile0 entry 0
```

```
#sql { SELECT LASTNAME  , FIRSTNME  , SALARY   FROM DSN8710.EMP  WHERE SALARY BETWEEN  ?
AND  ?   ORDER BY LASTNAME, FIRSTNME    };
line number: 49
PREPARED_STATEMENT executed via EXECUTE_QUERY
role is QUERY
descriptor is null
contains 2 parameters
1. mode: IN, java type: java.math.BigDecimal (java.math.BigDecimal),
   sql type: NUMERIC, name: min, marker index: 90
2. mode: IN, java type: java.math.BigDecimal (java.math.BigDecimal),
   sql type: NUMERIC, name: max, marker index: 98
result set Type is POSITIONED_RESULT
result set name is com.ibm.itso.sg246435.sqlj.Hello$EmployeeIterator
contains 3 result columns
1. mode: OUT, java type: java.lang.String (java.lang.String),
   sql type: VARCHAR, name: getCol1, marker index: -1
2. mode: OUT, java type: java.lang.String (java.lang.String),
   sql type: VARCHAR, name: getCol2, marker index: -1
3. mode: OUT, java type: java.math.BigDecimal (java.math.BigDecimal),
   sql type: NUMERIC, name: getCol3, marker index: -1

---------- EntryInfo custom data ----------                                         3
 DB2 Statement Type: 231
 Needs runtime describe: false
 Section: com.ibm.db2.jcc.sqlj.StaticSection@13b08d92
  Section number: 1
 DB2 parameterMetaData:com.ibm.db2.jcc.SQLJColumnMetaData@b124d92
 Parameter 1:                                                                       4
   name:null
   label:null
   nullable:true
   sqlType:485
   precision:9
   scale:2
   ccsid:0
   columnLength:0
   tableName:
 Parameter 2:
   name:null
   label:null
   nullable:true
   sqlType:485
   precision:9
   scale:2
   ccsid:0
   columnLength:0
   tableName:
Query-Related Information:                                                          5
 cursorName: DB2JCCCURSOR1
 holdability: false
 cursorType: 1003
 resultSetConcurrency: 1007
  dynamicUpdate section: null
 DB2 resultSetMetaData:com.ibm.db2.jcc.SQLJColumnMetaData@110c8d92
 Parameter 1:                                                                       6
   name:LASTNAME
   label:null
   nullable:false
   sqlType:448
   precision:0
```

```
    scale:0
    ccsid:37
    columnLength:15
    tableName:<not described>
  Parameter 2:
    name:FIRSTNME
    label:null
    nullable:false
    sqlType:448
    precision:0
    scale:0
    ccsid:37
    columnLength:12
    tableName:<not described>
  Parameter 3:
    name:SALARY
    label:null
    nullable:true
    sqlType:485
    precision:9
    scale:2
    ccsid:0
    columnLength:0
    tableName:<not described>
Positioned Update/Delete-specific Information:
 forUpdateCursorNames: null

====================================================

---------- DB2 ProfileData custom data ----------                          7
Package Name: HELLO
Package Version: null
Package Collection: SG246435
Is Default Collection: false
Package Consistency Token, hex format: 4241635543474870
Package Consistency Token, character format: BAcUCGHp
```

Notes on Example 9-7 on page 169:

1. Now we see that there is one customization in the profile.
2. We can immediately see that it is a DB2 customization.
3. Everything beyond this point is part of the customization.
4. There is a description of all the input host variables: Their data type, nullability, code page and length.
5. There is information about the holdability of the cursor.
6. There is a also a description of all the (output) columns in the SELECT list with similar information as for the input host variables, such as their data type, nullability, code page and length.
7. The last section provides information about the DB2 packages that correspond with this serialized profile. It shows the package and collection name, and the consistency token.

Executing the sqlj application using statically bound statements

Now we are ready to execute our program. The command, as well as the results are identical of course (Example 9-8 on page 172). You can use the same means as described in

“Verifying that your application uses static SQL” on page 162 to check whether the program actually uses static SQL.

Example 9-8 Executing the sqlj program using static SQL

```
C:\Test SQLJ\SG24-6435>java com/ibm/itso/sg246435/sqlj/Hello
jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y bart b1ngo 30000 50000

GEYER, JOHN: $40175.00
HEMMINGER, DIAN: $46500.00
KWAN, SALLY: $38250.00
LUCCHESI, VINCENZO: $46500.00
PULASKI, EVA: $36170.00
STIEN, IRVING: $32250.00
THOMPSON, MICHAEL: $41250.00
```

9.6.3 Running your sqlj program locally on a DB2 for z/OS system

To demonstrate the magic of this even better, we now demonstrate that you can just take the work that you did on your workstation, either manually or using WSAD, ship it to the mainframe and run your Java program there, just like that.

FTP the class files and the .ser file to z/OS

You can use the WSAD export facility, or if your program is just in your workstation file system, use plain FTP, to send the following files in binary to your z/OS HFS file system.

```
Hello$EmployeeIterator.class
Hello.class
Hello_SJProfileKeys.class

Hello_SJProfile0.ser
```

Note that we do not ship the source programs, as there is no need to retranslate, or recompile them on the z/OS system.

Run the program just like that on z/OS

Provided your system is set up correctly (correct CLASSPATH and such), you can now just run the program from USS on z/OS, like this:

```
java com/ibm/itso/sg246435/sqlj/Hello jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y bart
 b1ngo 30000 50000
```

Note that we use the exact same command that we did when running it from the Windows command prompt.

Also note that we use the `jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y` URL to address the DB2 for z/OS subsystem. However, using this URL format means that we use the Type 4 driver, using DRDA to talk to the local DB2 system. Although it works fine, performance-wise this is not the best solution.

So instead of using the Type 4 driver, we want to use the Type 2 driver. Nothing is easier than that. We just change the URL to `jdbc:db2:DB7Y`, and issue the following command:

```
java com/ibm/itso/sg246435/sqlj/Hello jdbc:db2:DB7Y bart b1ngo 30000 50000
```

And the program still runs, this time using the Type 2 driver, using the RRS attachment to talk to the local DB2 system.

Note: For things to work as they are described here, you need set up the system properly. This is described in more detail in Chapter 5, “Setup” on page 57.

The other thing to note is that the URL has to be external to the program itself, either through the use of a parameter, as in our example, or through an external property or an external DataSource definition.

9.6.4 In summary

As shown in the previous paragraphs, development and deployment of sqlj programs is very easy and could be done in the following way:

1. Users develop sqlj programs on a workstation platform using a nice integrated development environment like WSAD, and do some initial testing of their database calls against a DB2 database on z/OS, via dynamic SQL through JDBC (running uncustomized).
2. In the next phase, when the program is more or less ready, the serialized profile can be customized from the workstation, through WSAD if you want, and further testing can be done from the workstation against the DB2 for z/OS system, using static SQL this time.
3. When the program is ready to be taken into production, you only have to copy (for example, using FTP) the class files and the (customized) serialized profiles (in binary) from your workstation over to the mainframe platform and, for example, deploy them in your WebSphere for z/OS environment, and run a BIND COPY operation to bind your packages from the test system into the production system, and you are ready to go. Note that the taking into production can easily be set up to be implemented through a software that normally handles your change and version control environment.

10

SQLJ tutorial and reference

This chapter provides a tutorial and reference to SQLJ syntax and usage. It is not meant to be exhaustive. For a detailed description of the individual SQLJ constructs, please refer to *Application Programming Guide and Reference for Java™*, SC26-9932.

We discuss the following:

- The basic syntax of SQLJ statements
- Host variables and expressions
- Null values
- Data type mapping
- Queries, iterators, and the assignment statement
- Connection declarations
- Execution contexts
- Interoperability between JDBC and SQLJ

10.1 The basic syntax of SQLJ statements

There are four different forms of SQLJ statements:

- Executable statements
- Iterator declarations
- Assignment statements
- Connection contexts

We discuss each of these in the following sections.

10.1.1 Executable statements

This is the simplest SQLJ construct. You use an executable statement to perform database operations, such as:

- Inserting, modifying and deleting data (Data Manipulation Language (DML) statements)
- Creating database objects (Data Definition Language (DDL) statements)
- Control transactions (COMMIT and ROLLBACK statements)
- Calling stored procedures

An execute statement can appear anywhere a Java statement can appear. It begins with the token #sql (as does any other SQLJ statement), followed by an optional connection context specification and the SQL statement text, enclosed in curly braces, and is terminated by a semicolon:

```
#sql [connCtx] { SQL statement };
```

When you omit the [connCtx] clause:

```
#sql { SQL statement };
```

The statement is executed under the current *default connection context*. See "Connection contexts" on page 188 for why this is not good practice.

SQL statement can be almost any DB2 statement that can be statically prepared. Refer to *Application Programming Guide and Reference for Java™*, SC26-9932, for the full list of valid SQL statements.

Additionally, there is a special statement to set the isolation level for the current transaction:

```
#sql [ctx] { SET TRANSACTION ISOLATION LEVEL level };
```

Where *level* is one of the levels listed in Table 10-1.

Table 10-1 SQLJ isolation levels and their corresponding DB2 isolation levels

SQLJ isolation level	DB2 isolation level
READ UNCOMMITTED	UR (uncommitted read)
READ COMMITTED	CS (cursor stability)
REPEATABLE READ	RS (read stability)
SERIALIZABLE	RR (repeatable read)

Note: You can set the isolation level only at the beginning of a transaction, that is, before performing any other SQL operation.

The default isolation level for the DB2 JDBC drivers is READ COMMITTED (CS).

10.1.2 Iterator declarations

You use an iterator declaration to declare an SQLJ iterator class.To use an iterator, you assign the result of a SELECT statement to an instance of an iterator class, as discussed in “Queries, iterators, and the assignment statement” on page 182.

SQLJ iterators come in two different flavors: *Positioned iterators* and *named iterators*. The two kinds of iterators are implemented as different Java types and cannot be used interchangeably.

In this section, we only discuss *iterator declarations*. “Using positioned iterators” on page 182 and “Using named iterators” on page 183 explain how to *use* positioned and named iterators, respectively.

Positioned iterators

In its simplest and most common form, the declaration of a *positioned iterator* looks like this:

```
#sql modifiers iterator ClassName (type_list);
```

Where:

- *modifiers* is an optional list of Java class declaration modifiers (such as public, static, final, and synchronized).
- *ClassName* is a legal Java class name.
- *type_list* is a comma-separated list of Java types (which can be both primitive types such as int, or reference types such as String).

For example, to declare a positioned iterator for a result set of CHARACTER, INTEGER NOT NULL and TIMESTAMP:

```
#sql iterator MyPositionedIterator(String, int, Timestamp);
```

For each iterator declaration, the SQLJ translator generates a corresponding Java class. Therefore, an iterator declaration is allowed only where declaring a Java class is allowed.

If you do not specify a modifier, the generated iterator class will have default visibility, that is, it is visible only to classes in the same package. If you want to use it in other packages, you have to add the `public` modifier before the iterator keyword:

```
#sql public iterator ClassName(Type list);
```

However, the Java language specification mandates that you cannot have two top-level (that is, non-nested) public classes in the same source file. Therefore, if you declare your iterator `public`, this declaration must either be contained in a separate source file (whose base name must be the same as the iterator class name), or you have to declare it as an inner class (see Example 10-1).

Example 10-1 Iterator declaration as an inner class

```
public class MyClass {
  #sql public static iterator MyPositionedIterator(...);
  ...
```

```
}
```

Named iterators

A *named iterator* declaration looks very similar to a positioned iterator declaration, except it declares not only type names but also column names:

```
#sql modifiers iterator ClassName(name_list);
```

Where:

- *modifiers* Is an optional list of Java class declaration modifiers (such as `public`, `static`, `final`, and `synchronized`).
- *ClassName* Is a legal Java class name.
- *name_list* Is a comma-separated list of a Java type name followed by a Java identifier.

For example:

```
#sql iterator MyNamedIterator(String name, int edLevel, Timestamp dateHired);
```

The names in the *name_list* must match (ignoring case) the names of the columns you want to retrieve in a SELECT statement using the iterator (see “Using named iterators” on page 183 for more details).

Modifying iterator behavior using the implements and with clauses

In the previous sections, we only showed the basic form of iterator declarations. There are two optional clauses in the iterator declaration statement that are used for advanced features.

Feel free to skip this section on first reading, and refer back in the discussion of positioned UPDATE and DELETE, and holdable iterators, later in this chapter.

In its most general form, an iterator declaration looks like this:

```
#sql modifiers iterator ClassName implements interface_list with (keyword_with_list)
  (column_list);
```

Where:

- *interface_list* is a comma-separated list of Java interface names, just like in a Java class declaration.
- *keyword_with_list* is a comma-separated list of keyword-value pairs:

  ```
  keyword1=value1, keyword2=value2, ...
  ```

- *column_list* is either a list of column types (for positioned iterators), or column types and column names (for named iterators), as described above.

The `implements` *`interface_list`* clauses cause the generated iterator class to implement the interfaces listed in the clause, and may also cause the SQLJ translator to generate additional methods in the iterator class. There are two interfaces predefined by SQLJ:

- sqlj.runtime.ForUpdate, which marks the iterator as updateable
- sqlj.runtime.Scrollable, which marks the iterator as scrollable

An implementation is free to support additional interfaces.

> **Important:** You must code the fully qualified interface name—using an unqualified name does *not* work, even when you add an import statement for the interface. This is currently a limitation, but it will be addressed in future versions.

The optional *with clause* causes a public static final variable (in other words, a constant) of the appropriate type and the supplied value to be inserted into the generated iterator class declaration. There are three keywords predefined by the SQLJ standard:

- `sensitivity` (value must be one of `sqlj.runtime.ResultSetIterator.SENSITIVE`, `sqlj.runtime.ResultSetIterator.INSENSITIVE`, or `sqlj.runtime.ResultSetIterator.ASENSITIVE`)

  Controls whether the iterator is aware of changes to the underlying base table. For details, see the *Application Programming and SQL Guide*, SC26-9933.

- `holdability` (value must be `true` or `false`)

  See “Holdable iterators” on page 186 for more details.

- `updateColumns` (value must be a comma-separated list of column names)

  See “Positioned UPDATE and DELETE” on page 186 for more details.

- `dynamic` (value must be true or false)

  Controls whether a sensitive scrollable cursor is dynamic. A true value only has an effect on a server that supports dynamic sensitive scrollable cursors, like DB2 for z/OS Version 8.

In addition, an SQLJ implementation may accept additional keywords, allowing for vendor-specific extensions.

10.2 Host variables and expressions

You use host variables and host expressions to pass arguments to SQL statements. A *host variable* is simply a Java identifier (which could refer to a method parameter, a local variable, or a field). Host variables are prefixed with a colon (`:`).

Unlike other languages with embedded SQL support, SQLJ also supports *host expressions*. A host expression is any legal Java expression that yields a value. Like host variables, host expressions are prefixed by a colon, but must additionally be enclosed in parentheses.

10.3 Null values

SQL supports the distinct value “NULL” to denote the absence of a value in a column. Note that the NULL value is not the same as the numeric value 0 (that is, zero). Unless a column is declared NOT NULL, you can insert NULL values into, and your queries must be prepared to receive NULL values from, that column.

In other languages with embedded SQL support, such as COBOL or C, you have to use an additional construct called a NULL indicator. Java, however, does have a distinct null value for reference types—the Java null is equivalent to the SQL NULL.

Whenever you want to retrieve data from a column with NULLs allowed, make sure that the corresponding host variable is of reference type (as opposed to the primitive types, for example, `int` and `boolean`). That is, use the wrapper classes in the `java.lang` package.

For example, consider the following code snippet:

```
short edlevel;
#sql [ctx] {
  SELECT EDLEVEL
    INTO :edlevel
```

```
    FROM DSN8710.EMP
   WHERE ...
};
```

Since the EDLEVEL column in the DSN8710.EMP sample table is declared nullable, the program will fail with an `SQLNullException` (a subclass of `SQLException`) when a NULL value is encountered. To fix this, use the wrapper class corresponding to the primitive type:

```
Short edlevel;
#sql [ctx] {
  SELECT EDLEVEL
    INTO :edlevel
    FROM DSN8710.EMP
   WHERE ...
};
if (edlevel == null)
  System.out.println("Unknown education level");
else
  System.out.println("Education level: " + edlevel);
```

Alternatively, you could still use a primitive and handle the SQLNullException:

```
short edlevel;
try {
  #sql [ctx] {
    SELECT EDLEVEL
      INTO :edlevel
      FROM DSN8710.EMP
     WHERE ...
  };
  System.out.println("Education level: " + edlevel);
} catch (SQLNullException e) {
  System.out.println("Unknown education level");
}
```

But this is considered bad programming style in most cases since exceptions should not be used for regular control flow. Use this technique only when you do not really expect the column to be NULL although it technically could be (for example, due to some business constraint that cannot be expressed in DDL). In all other cases, using the wrapper type is the cleaner solution.

10.4 Data type mapping

Your choice of Java data types can affect performance, because DB2 picks better access paths when the data types of your Java variables map closely to the DB2 data types. Also, unlike other languages with SQL support, JDBC and SQLJ provide data types to map to the SQL date/time column types.

The “best” mappings of Java to DB2 data types are summarized in Table 10-2 on page 181. When the Java type column gives two Java types, use the first one (the primitive type) for NOT NULL columns and the second one (the corresponding wrapper type) for nullable columns.

Table 10-2 Best mappings of Java to DB2 data types

Java type	DB2 type	Comments
boolean Boolean	SMALLINT	DB2 has no exact equivalent, but SMALLINT is the best match. A zero value denotes false; any non-zero value denotes true.
byte Byte	SMALLINT	DB2 has no exact equivalent, but SMALLINT is the best match.
short Short	SMALLINT	
int Integer	INTEGER	
long Long	DECIMAL(19,0)	DB2 has no 64-bit integer type. However, a DECIMAL column with precision 19 can hold all long values.
java.math.BigInteger	DECIMAL(19,0)	
float Float	REAL	
double	DOUBLE or FLOAT	Somewhat confusingly, FLOAT is a synonym for DOUBLE in DB2.
java.math.BigDecimal	DECIMAL(p,s)	p = precision, s = scale. For example, a DECIMAL(5,2) column has a range of 000.00 to ±999.99.
java.lang.String	CHAR(n)	Declares a fixed-width column of length n, where $1 \leq n \leq 255$. The value will be padded with trailing blanks if necessary.
	VARCHAR(n)	Declares a variable-width column of maximum length n. No padding occurs. Certain restrictions apply if n > 255 (see *DB2 Universal Database for OS/390 and z/OS SQL Reference*, SC26-9944).
	GRAPHIC(n)	Declares a fixed-width DBCS column of length n, where $1 \leq n \leq 255$. The value will be padded with trailing blanks if necessary.
	VARGRAPHIC(n)	Declares a variable-width DBCS column of length n, where $1 \leq n \leq 255$. The value will be padded with trailing blanks if necessary.
byte[]	CHAR(n) FOR BIT DATA	
	VARCHAR(n) FOR BIT DATA	
java.sql.Date	DATE	
java.sql.Time	TIME	
java.sql.Timestamp	TIMESTAMP	
java.sql.Blob	BLOB(n)	

Java type	DB2 type	Comments
java.sql.Clob	CLOB(n)	
	DBCLOB(m)	

10.5 Queries, iterators, and the assignment statement

To retrieve the result set of a SELECT statement, you assign that result set to an instance of a previously declared iterator class. You then iterate over the result set, fetching one row at a time, until all result set rows have been retrieved.

Exactly how you iterate over the result set and how you retrieve the values depends on whether you are using positioned iterators or named iterators. We explain both variants in "Using positioned iterators" on page 182 and "Using named iterators" on page 183, respectively.

Normally, iterators are automatically closed when the current transaction is committed. When you need an iterator to remain open across transaction boundaries, you have to declare them holdable. "Holdable iterators" on page 186 shows how to do this.

You can also update or delete the row an iterator is currently positioned on. This is explained in "Positioned UPDATE and DELETE" on page 186.

10.5.1 Using positioned iterators

To use a positioned iterator, assign it the result from a SELECT statement. Example 10-2 shows the technique.

Example 10-2 Assigning the result set of a query to an iterator

```
#sql iterator MyPositionedIterator(String, int, Timestamp);
.
.
.
MyPositionedIterator iter;
#sql [ctx] iter = {
  SELECT NAME
       , EDLEVEL
       , DATEHIRED
    FROM DSN8710.EMP
   WHERE ...
};
```

The number of the columns in the iterator declaration must match the number of columns in the select list; otherwise, the SQLJ translator reports an error. (However, the error will be reported against the FETCH statement, not the SELECT statement or the iterator.) Also, the data types in the iterator declaration must be compatible with the column types. The SQLJ profile customizer will check and report any data type mismatches, if online checking is enabled.

You then retrieve the rows in the result set by executing FETCH statements, usually in a loop. The column values will be assigned to the corresponding host variables in the INTO list. To find out if all the rows have been retrieved, you call the `endFetch()` method of the iterator. This technique is demonstrated in Example 10-3 on page 183.

Example 10-3 Fetching result set rows into host variables

```
String    name = null;
int       edLevel;
Timestamp dateHired = null;

try {
  while (true) {
    #sql {
      FETCH :iter
       INTO :name
          , :edLevel
          , :dateHired
    };
    if (iter.endFetch())
      break;
    System.out.println(name + '\t' + edLevel + '\t' + dateHired);
  }
} finally {
  iter.close();
}
```

Note that `endFetch()` reports the result of the last attempt to retrieve a row, returning false when a row had been fetched successfully, and true if no more rows could be retrieved. This implies that *it returns a meaningful result only after a FETCH has been performed*. The following snippet (Example 10-4) is *not* guaranteed to work as expected.

Example 10-4 Wrong usage of endFetch() - Do not try this at home

```
// WRONG -- will not work as expected
try {
  while (!iter.endFetch()) {
    #sql {
      FETCH :iter
       INTO :name
          , :edLevel
          , :dateHired
    };
    System.out.println(name + '\t' + edLevel + '\t' + dateHired);
  }
} finally {
  iter.close();
}
```

Either the loop will not be entered at all (since no FETCH has been performed initially, `endFetch()` may return true), or the last result row will be printed twice (a FETCH operation leaves the host variables unchanged when there is no more row to retrieve).

10.5.2 Using named iterators

While the declaration of a named iterator looks very similar to a positioned iterator, the usage is quite different. You do not use the FETCH statement to retrieve the next row and assign the column values automatically to host variables. Rather, you use the `next()` method of the iterator class to retrieve the next row, and you use generated accessor methods of the iterator class to retrieve the individual columns. The `next()` method returns false when there is no next row to retrieve.

Unlike with named iterators (see “Using positioned iterators” on page 182), the `next()` method can be used as the control expression of the `while` statement since it tries to fetch a row, and returns false there were no more rows in the result set.

Example 10-5 Using a named iterator

```
#sql iterator MyNamedIterator(String lastName, int edLevel, Timestamp hireDate);
.
.
.
MyNamedIterator iter;
#sql [ctx] iter = {
  SELECT LASTNAME
       , EDLEVEL
       , HIREDATE
    FROM DSN8710.EMP
   WHERE ...
};
while (iter.next()) {
  System.out.println(iter.lastName()
                     + ‘\t’ + iter.edLevel()
                     + ‘\t’ + iter.hireDate());
}
```

Although the named iterator variant may look more familiar, especially to seasoned JDBC programmers, we suggest that you use positioned iterators instead, since they are slightly more efficient. Named iterators actually use positioned iterators under the cover, with an additional hash table to map column names to column positions.

10.5.3 SQLJ iterators versus cursors

If you are familiar with embedded SQL in other languages, such as COBOL or C, you may have noted that SQLJ iterators differ in one important respect from SQL cursors. (If you are not, and you have no idea what we are talking about, feel free to skip this section.)

An SQLJ iterator is a regular Java object that can be passed, for example, to other methods, even in different classes. In other words, the SELECT statement populating the iterator and the FETCH statement to retrieve the result rows may well be in different source files. An SQL cursor, on the other hand, can only be used in the source file where it was declared.

This is illustrated by the fact that, in a FETCH statement, you refer to a Java iterator by a host variable reference (as indicated by a colon), whereas in other languages you refer to cursors by their declared names.

Another significant difference is that the declaration of an SQL cursor includes the SELECT statement; that is, you have to declare one cursor per SELECT statement. An iterator declaration in SQLJ is different; it only specifies the column types (and names, in the case of named iterators). Therefore, you can assign the result of any SELECT statement to an iterator instance as long as the columns in the SELECT list match the iterator.

We demonstrate this in Examples 10-6 and 10-7.

Example 10-6 Cursor declaration includes SELECT statement

```
*
* COBOL people: this is only pseudocode.
* Java people: this is the only COBOL sample in this book, so keep reading.
*
EXEC SQL
```

```
  DECLARE EMP-MIN-MAX CURSOR FOR
   SELECT LASTNAME, HIREDATE
     FROM DSN8710.EMP
    WHERE SALARY BETWEEN :MIN AND :MAX;

EXEC SQL
  DECLARE EMP-ALL CURSOR FOR
   SELECT LASTNAME, HIREDAE
     FROM DSN8710.EMP;

...

EXEC SQL OPEN EMP-MIN-MAX;
PERFORM WITH CHECK AFTER UNTIL SQLCODE IS EQUAL TO 100
  EXEC SQL FETCH EMP-MIN-MAX
            INTO :LASTNAME
               , :HIREDATE;
  ...
END-PERFORM.
EXEC SQL CLOSE EMP-MIN-MAX;

EXEC SQL OPEN EMP-ALL;
PERFORM WITH CHECK AFTER UNTIL SQLCODE IS EQUAL TO 100
  EXEC SQL FETCH EMP-ALL
            INTO :LASTNAME
               , :HIREDATE;
  ...
END-PERFORM.
EXEC SQL CLOSE EMP-ALL;
```

Let us now look at a similar example using SQLJ.

Example 10-7 Iterator declaration does not include SELECT statement

```
#sql iterator EmployeeIterator(String, Date);
...
EmployeeIterator iter;

#sql [ctx] iter = {
  SELECT LASTNAME, HIREDATE
    FROM DSN8710.EMP
   WHERE SALARY BETWEEN :min AND :max
};
...
while (true) {
  #sql { FETCH :iter INTO :lastname, :hiredate };
  if (iter.endFetch()) break;
  ...
}
iter.close();

#sql [ctx] iter = {
  SELECT LASTNAME, HIREDATE
    FROM DSN8710.EMP
};

while (true) {
  #sql { FETCH :iter INTO :lastname, :hiredate };
  if (iter.endFetch()) break;
  ...
```

```
}
iter.close();
```

As you can see, we use the same iterator for both SQL statements.

10.5.4 Holdable iterators

Normally, an iterator is automatically closed during a COMMIT operation. If you want your iterator to remain open after a COMMIT, you must declare the corresponding iterator class as holdable, using the `with` clause discussed in “Modifying iterator behavior using the implements and with clauses” on page 178:

```
#sql iterator MyIterator with (holdability=true) (String, int, Timestamp) ;
```

Be aware, however, that holdable iterators can impact performance and concurrency since resources and locks held by such iterators will not be released until you close them.

During a ROLLBACK operation, however, DB2 closes all iterators, including those that were declared holdable.

10.5.5 Positioned UPDATE and DELETE

Normally, you update or delete rows by doing what is called a *searched* UPDATE or DELETE. That is, you specify which rows to be updated or deleted through a WHERE clause. For example, to give a bonus of $500.00 to all employees who work in department E21, you code:

```
#sql [ctx] {
  UPDATE DSN8710.EMP
     SET BONUS = 500.00
   WHERE WORKDEPT = 'E21'
};
```

In some situations, however, you may want to change columns as you iterate through the result set of a query. For example, in an interactive application, the manager may decide what bonus to give each employee while browsing through the list of all employees in his department.

You can update or delete the row on which an iterator is currently positioned, by executing the positioned UPDATE or DELETE statements, respectively.

```
#sql [ctx] {
  UPDATE DSN8710.EMP
     SET BONUS = :bonus
   WHERE CURRENT OF :iter
};

#sql [ctx] {
  DELETE FROM DSN8710.EMP
   WHERE CURRENT OF :iter
};
```

However, you have to declare the corresponding iterator as *updateable*. You do this by adding the clause `implements sqlj.runtime.ForUpdate` after the iterator class name (again, refer to “Modifying iterator behavior using the implements and with clauses” on page 178 for a discussion of the `implements` clause):

```
#sql iterator MyIter implements sqlj.runtime.ForUpdate (columnList);
```

By default, all columns in the result table will then be updateable through this iterator (even columns that are not in the SELECT list). Optionally, you can supply a list of columns you want to update, using the `with` clause:

```
#sql iterator MyIter implements sqlj.runtime.ForUpdate
  with (updateColumns="COL1, COL2") (columnList);
```

If you know beforehand which columns you are going to update, you should specify this clause, since this normally leads to a more efficient program.

Let us look at Example 10-8 below.

Example 10-8 Positioned UPDATE

```
package com.ibm.itso.sg246435;

import java.math.BigDecimal;

import java.sql.SQLException;

public class UpdateBonus {

  #sql public static iterator EmployeeIterator                              1
            implements sqlj.runtime.ForUpdate
            with (holdability=true, updateColumns="BONUS")                  2
  (
      String
  );

  public static void updateBonus(String dept, BigDecimal bonus) throws SQLException {
    EmployeeIterator empiter;
    #sql [ctx] empiter = {
      SELECT LASTNAME || ', ' || FIRSTNME || ' ' || MIDINIT
        FROM DSN8710.EMP
       WHERE WORKDEPT = :dept
    };
    try {
      String name = null;
      int numrows = 0;                                                      3
      while (true) {
        #sql { FETCH :empiter INTO :name };
        if (empiter.endFetch()) break;
        #sql [ctx] {
          UPDATE DSN8710.EMP                                                4
             SET BONUS = :bonus
           WHERE CURRENT OF :empiter
        };
        System.out.println(name);
        if (++numrows % 100 == 0) {                                         5
           #sql [ctx] { COMMIT };
        }
      }
      #sql [ctx] { COMMIT };                                                6
    } finally {
      empiter.close();                                                      7
    }
  }
  ...
}
```

Notes on Example 10-8 on page 187:

1. Declares the iterator class. The iterator must implement `sqlj.runtime.ForUpdate` in order to use it in a positioned UPDATE statement.
2. In this example, we combine the `holdability` and `updateColumns` attributes.
3. Counts the number of rows changed.
4. This is the positioned UPDATE statement. It modifies the row the iterator is currently positioned on. Note that we update a column that is not part of the SELECT list.
5. Commits every 100 rows.
6. Commits any remaining uncommitted changes.
7. It is good practice to close the iterator in a finally block, so the iterator will be closed even if an exception is thrown. This is especially important for holdable iterators.

10.5.6 Calling stored procedures

To call a stored procedure, you use the CALL statement:

```
#sql { CALL procname(:arg1, :arg2, ...) };
```

Since stored procedures have input, output, and input/output variables, it is possible that the values of the host variables are changed by invoking the procedure.

10.6 Connection contexts

Each SQLJ statement runs in a given *connection context*, which is a Java object that encapsulates a database connection (similar to a JDBC `Connection` object), and manages the use of serialized profiles in the program.

10.6.1 Setting up and using an implicit connection context

Unless you specify it on the SQLJ statement, each statement will be executed under an implicit connection context, which in turn is usually an instance of class `sqlj.runtime.ref.DefaultContext`. This class is part of the SQLJ runtime library.

> **Important:** Except maybe in stand-alone programs (that do not share a JVM with other applications) using an implicit context is strongly discouraged. “Why the connection context is important” on page 189 explains why.

Depending on the environment your program is running in, a default connection context may or may not have been set up for you. In fact, the SQLJ runtime implementation tries to create a default context by doing a JNDI lookup for `jdbc/defaultDataSource`, so when you are in an environment that has access to a JNDI namespace, and a DataSource under that name has been found, the default context is set up automatically.

To set up a default connection context yourself, use one of the following two alternatives:

- Creating a Connection and a DefaultContext in one step:

  ```
  DefaultContext ctx = new DefaultContext(url, userid, password, autoCommit);
  DefaultContext.setDefaultContext(ctx);
  ```

  Because in most cases you want to be in control of the commit scope of your application, make sure that autocommit is false, for example:

```
DefaultContext ctx = new DefaultContext("jdbc:db2os390sqlj:NEWYORK",
                                        "bart", "b1ngo", false);
DefaultContext.setDefaultContext(ctx);
```

- Creating a DefaultContext using an existing JDBC connection:

```
DefaultContext ctx = new DefaultContext(conn);
DefaultContext.setDefaultContext(ctx);
```

10.6.2 Why the connection context is important

As we said above, an SQLJ statement that does not explicitly specify a connection context runs under a default context (which you could have established yourself, or which may have been set up by the environment in which your application is running). This saves you some typing since you do not need to code the [ctx] clause on each statement.

However, we do *not* recommend using the default context, except maybe in stand-alone programs that do not share a JVM with other applications.

The default context is a static variable of class `sqlj.runtime.ref.DefaultContext` (the `setDefaultContext()` method is used to modify it). Now suppose your application is running in a multithreaded environment, such as WebSphere Application server. There are two obvious problems with using the default context:

- First, you potentially share a single database connection with other threads, which can create a throughput bottleneck.
- Second, there is no guarantee that, between the time your application sets the default context and then executes using an implicit context, another thread has not changed the value for the default context. After all, the default context is a global variable you are sharing with all threads running in the same JVM.

Strictly speaking, it is not necessary to declare a separate context class, using the `#sql context` clause. You can use the `DefaultContext` class supplied with the SQLJ runtime library, as long as you do not call `setDefaultContext()` and use an implicit context. For maximum portability, however, the SQLJ standard recommends that you always declare (and use) explicit context, since the specification of the default context is implementation-defined.

There are situations, however, where it is mandatory to declare your own context classes. This is explained in “Using more than one context class” on page 194.

We describe how to declare and use an explicit context in the following sections.

10.6.3 Declaring a context class

The declaration of an SQLJ context class is similar to an SQLJ iterator declaration:

```
#sql modifiers context ContextClassName [ with (keyword=value, ...) ]
```

Where:

- *modifiers* is a list of Java modifiers as explained in “Iterator declarations” on page 177.
- *ContextClassName* is a valid Java class name.
- The optional *with* clause causes constants to be inserted in the generated connection context class, as explained in “Modifying iterator behavior using the implements and with clauses” on page 178, allowing for vendor-specific extensions. (There are also some predefined keywords, most notably `dataSource`, which is explained in “Creating an instance of the context class” on page 190.)

As with iterator declarations, the context declaration is translated into a class declaration by the SQLJ translator. Therefore, a context declaration is only allowed where a Java class declaration would be allowed.

10.6.4 Creating an instance of the context class

To set up a context, you create an instance of the context class and assign it to a Java variable. The context class has several constructors:

- public MyCtx(java.sql.Connection conn)
- `public MyCtx(sqlj.runtime.ConnectionContext other) throws java.sql.SQLException`
- public MyCtx(String url, String user, String password, boolean autoCommit) throws java.sql.SQLException
- public MyCtx(String url, java.util.Properties info, boolean autoCommit) throws java.sql.SQLException
- public MyCtx(String url, boolean autoCommit) throws java.sql.SQLException

The first of those constructors accepts a JDBC Connection object; the second uses an existing connection context. The others look very similar to the methods of the `java.sql.DriverManager` class (and in fact create a `Connection` object using these parameters), except that, additionally, you specify whether or not you want automatic commit enabled for this context.

If you specify the `with(dataSource=”`*`jndiName`*`”)` clause, a no-argument constructor will also be generated, which creates the connection via a DataSource object obtained by the given JNDI name.

10.6.5 Specifying which connection instance to use

To specify which connection instance to use for an SQLJ statement, you specify the name of the variable holding the reference to that instance, enclosed in square brackets, after the `#sql` token. For example:

```
#sql context MyCtx;
...
MyCtx ctx;
ctx = new MyCtx(url, username, password, false); // false means no auto commit
...
#sql [ctx] { COMMIT };
```

> **Note:** The only exception is the `FETCH INTO` statement, which does not need an explicit connection clause (in fact, the translator issues a warning if you code it).

This is demonstrated in Example 10-9. We create two connection context instances, and then start three threads; one of them updates a row in the database, the other two retrieve a column of that row. One of those, in turn, uses the connection context that had also been used by the updating thread; the other uses another context.

Example 10-9 Using different connection contexts

```
package com.ibm.itso.sg246435.sqlj;

import java.sql.*;
import sqlj.runtime.ref.*;
import java.math.BigDecimal;
```

```
public class ConnectionContextTest {

  private static final String url = "jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y";
  private static final String user = "bartr1";
  private static final String password = "secret";
  private static final String empno = "000130";

  #sql static context Ctx;                                                1

  private static Ctx ctx1;                                                2
  private static Ctx ctx2;

  private static boolean updated = false;
  private static Object semaphore = new Object();

  public static void main(String[] args) {
    try {
      Class.forName("com.ibm.db2.jcc.DB2Driver");

      // Set up connection contexts with autocommit disabled
      ctx1 = new Ctx(url, user, password, false);                         3
      ctx2 = new Ctx(url, user, password, false);

      new Thread() {                                                      4
        public void run() { updateOnCtx1(); }
        public String toString() { return "Thread 1 [ctx1]"; }
      }.start();

      new Thread() {                                                      5
        public void run() { selectOnCtx1(); }
        public String toString() { return "Thread 2 [ctx1]"; }
      }.start();

      new Thread() {                                                      6
        public void run() { selectOnCtx2(); }
        public String toString() { return "Thread 3 [ctx2]"; }
      }.start();

    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  private static void message(String message) {
    System.out.println(Thread.currentThread() + ": " + message);
  }

  private static void updateOnCtx1() {
    try {
      BigDecimal salary = null;
      message("SELECT");                                                  7
      #sql [ctx1] {
          SELECT SALARY
            INTO :salary
            FROM DSN8710.EMP
           WHERE EMPNO = :empno
      };
      message("Salary = " + salary);
```

```
        message("Updating row");                                                    8
        #sql [ctx1] {
        UPDATE DSN8710.EMP
           SET SALARY = SALARY + 300
         WHERE EMPNO = :empno
        };
        synchronized (semaphore) {                                                  9
          updated = true;
          semaphore.notifyAll();
        }
        message("Sleeping for 10 seconds");                                         10
        Thread.sleep(10000);
        message("Awake");
        message("Rollback");
        #sql [ctx1] { ROLLBACK };                                                   11
      } catch (Exception e) {
        message(e.toString());
      }
    }

    private static void selectOnCtx1() {
      try {
        BigDecimal salary = null;
        synchronized (semaphore) {                                                  12
          while (!updated)
            semaphore.wait();
        }
        message("SELECT");
        #sql [ctx1] {                                                               13
            SELECT SALARY
              INTO :salary
              FROM DSN8710.EMP
             WHERE EMPNO = :empno
        };
        message("Salary = " + salary);
      } catch (Exception e) {
        System.out.println(Thread.currentThread() + ": " + e);
      }
    }

    private static void selectOnCtx2() {
      try {
        BigDecimal salary = null;
        synchronized (semaphore) {
          while (!updated)
            semaphore.wait();
        }
        message("SELECT");
        #sql [ctx2] {                                                               14
            SELECT SALARY
              INTO :salary
              FROM DSN8710.EMP
             WHERE EMPNO = :empno
        };
        message("Salary = " + salary);
      } catch (Exception e) {
        message(e.toString());
      }
    }
  }
```

Notes on Example 10-9 on page 190:

1. Declare a connection context class…
2. … and two instances of that class.
3. Initialize the context instances. Each instance represents a separate database connection.
4. Start the thread which updates a database row.
5. Start the thread that selects on the same context instance as the updating thread.
6. Start the thread that selects on a different context instance.
7. First, the updating thread retrieves and prints the old value.
8. Now, it updates one row in the database.
9. We want the two other threads to not start querying the database before the update has been performed. To ensure this, we use a boolean flag and a monitor object. This is a Java technique that has nothing to do with the database.
10. Next, the updating thread sleeps for 10 seconds, giving the two other threads the chance to query the database before rolling back.
11. Roll back the change.
12. This loop waits for Thread 1 to signal that the row has been updated (see 9).
13. Single-row SELECT on ctx1, which is the same context on which the update has been performed.
14. Single-row SELECT on a different context instance.

The output from running this example is shown in Example 10-10.

Example 10-10 Output from Example 10-9 on page 190

```
Thread 1 [ctx1]: SELECT                                          1
Thread 1 [ctx1]: Salary = 23800.00                               2
Thread 1 [ctx1]: Updating row                                    3
Thread 1 [ctx1]: Sleeping for 10 seconds                         4
Thread 2 [ctx1]: SELECT                                          5
Thread 3 [ctx2]: SELECT                                          6
Thread 2 [ctx1]: Salary = 24100.00                               7
Thread 1 [ctx1]: Awake                                           8
Thread 1 [ctx1]: Rollback                                        9
Thread 3 [ctx2]: Salary = 23800.00                               10
```

What happens is the following:

1. The updating thread first issues a SELECT…
2. … and retrieves the old value, 23800.00.
3. Next, it updates the row, obtaining an exclusive lock on the row (or page).
4. It now sleeps for 10 seconds, giving the other threads a chance to query the database.
5. The second…
6. … and third thread each issue their SELECT statements.
7. Thread 2, which uses the same context instance as Thread 1, immediately gets its result, seeing the updated (but uncommitted) value. Thread 3, on the other hand, has to wait for the exclusive lock on the updated row to be released.
8. After 10 seconds, Thread 1 wakes up…

9. … and rolls back the change, also releasing the lock on the modified row.
10. Now that Thread 1 released the lock, Thread 3 is unblocked and sees the old value of the column.

10.6.6 Using more than one context class

As explained before, it is not strictly necessary (but recommended) to declare your own context class—you could create and use an instance of class sqlj.runtime.ref.DefaultContext, as long as you do not use implicit contexts (by omitting the [ctx] clause).

However, there are two situations where it is mandatory to declare your own context classes:

- Your application connects to different database servers.

  Recall the role of the profile customizer, discussed in "The DB2 profile customizer" on page 154. What the customizer and binder do is to inspect each SQL statement in the .ser file, check it for correct syntax and authorization, and create a package in the database for the .ser file.

  Obviously, we cannot have a single .ser file in a program that connects to different servers. Rather, we need several of them, each one containing only the SQL statements meant for one particular database server, and we need to run the customizer multiple times, once for each .ser file.

- You use unqualified SQL, and use different schemas.

  This situation is similar to the one above. You must be able to distinguish between the different schema names, since the object names on the statements are unqualified. Without declaring separate context classes, the profile customizer cannot know against which schema to check your statements. Furthermore, the default schema name that applies when using unqualified SQL is recorded in the database package. Obviously, when using unqualified SQLJ, we must have several packages—one for each schema used by the program.

This is where the context class declaration comes into play. The SQLJ translator will create a separate serialized profile (.ser file) for each set of statements in the program that executes on a different context class. This allows subsequent customization and bind to occur against different locations for each set of statements. Context instances that will connect to the same database should be instances of the same context class, and context instances that will connect to different databases should be instances of different context classes.

10.6.7 Summary of ConnectionContext methods

Hereafter we give a brief summary of the different ConnectionContext methods that you have at your disposal.

- getConnection

  ```
  public java.sql.Connection getConnection()
  ```

  Returns the JDBC connection object associated with this connection context.

- close

  ```
  void close(boolean flag) throws SQLException
  ```

  Closes the connection context, releasing all database resources it currently uses.

  – If `flag == ConnectionContext.CLOSE_CONNECTION`, it also closes the underlying JDBC connection.

  – If `flag == ConnectionContext.KEEP_CONNECTION`, the JDBC connection remains open.

```
public void close() throws SQLException
```

Closes the connection context and also closes the underlying JDBC connection. Equivalent to `close(ConnectionContext.CLOSE_CONNECTION)`.

- isClosed

  ```
  public boolean isClosed()
  ```

  Returns `true` if this connection context is closed, otherwise returns `false`.

- getExecutionContext

  ```
  public sqlj.runtime.ExecutionContext getExecutionContext()
  ```

  Returns the default execution context for this connection context. See 10.7, “Execution contexts” on page 195 for a discussion of execution contexts and their relation to connection contexts.

10.7 Execution contexts

Each SQLJ statement is associated, either implicitly or explicitly, with an *execution context*. This is an instance of the `sqlj.runtime.ExecutionContext` class, which in turn is part of the SQLJ runtime.

The execution context allows you to:

- Control the execution of statements running under that context.

  For example, you can set a query timeout for SQLJ statements. If the timeout is exceeded, an `SQLException` is thrown (if supported by the SQLJ implementation).

- Retrieve information about the status of the last SQLJ statement executed under that context.

  For example, you can retrieve the number of rows affected by the last SQL statement.

- Cancel a statement currently executing under that context.

  The `cancel()` method allows you to cancel the operation currently running under a given execution context. Obviously, you can call this method only from a thread other than the one executing the statement.

  > **Restriction:** At the time of writing, query timeout and asynchronous cancel is supported on DB2 UDB Version 8 for Unix and Windows only. It will also be supported for Type 4 connections accessing a DB2 for z/OS Version 8.

- Execute statements in batch.

  Using the `setBatching()`, `setBatchLimit()`, and `executeBatch()` methods, you can execute several statements in a batching fashion, which may result in better performance, especially in a client/server environment.

Execution contexts are especially important in multithreaded applications sharing the same connection context. If a thread attempts to execute an SQL statement using an execution context that currently is in use by another thread, a `RuntimeException` will be thrown. Therefore, execution contexts may only be shared between threads if their usage is properly synchronized. This includes calling getter methods on the execution context. They must be protected by the same synchronized block as the operation, since otherwise there would be a potential race condition between threads, meaning that the information could have been overwritten by another thread in the meantime (Example 10-11 on page 196).

Example 10-11 Synchronizing access to the execution context

```
// WRONG -- warnings may have been overwritten by another thread
synchronized (execCtx) {
  #sql [connCtx, execCtx] { UPDATE ... };
}
SQLWarning warning = execCtx.getWarnings();

// CORRECT -- operation and access to execution context is guarded by same block
SQLWarning warning = null;
synchronized (execCtx) {
  #sql [connCtx, execCtx] { UPDATE ... };
  warning = execCtx.getWarnings();
}
```

> **Tip:** To avoid these problems altogether, we recommend that each thread should use an execution context of its own when sharing a common connection context.

However, each connection context has its own default execution context. Therefore, when you use a different connection context in each thread, there usually is no need to explicitly create execution contexts.

When you are using the same connection context in different threads, and you want to execute SQL statements in parallel, you can create additional execution contexts. You then specify on each SQLJ statement in which context you want this particular statement to be executed.

Unlike connection contexts, there is only one class for execution contexts—no construct is available to declare user-defined execution context classes. To create a new execution context, simply create a new instance of class `sqlj.runtime.ExecutionContext` and store the reference to that instance:

```
import sqlj.runtime.ExecutionContext;
...
ExecutionContext execCtx = new ExecutionContext();
```

To specify an execution context to be used for a particular SQLJ statement, use the following syntax:

```
#sql [connCtx, execCtx] { SQL statement };
```

It is also possible to only specify an execution context, in which case the default connection context will be used:

```
#sql [execCtx] { SQL statement };
```

Summary of ExecutionContext methods

Hereafter we give a brief summary of the different ExecutionContext methods that you have at your disposal.

- cancel

  `public void cancel() throws SQLException`

  Tries to cancel the SQL operation currently executing under this ExecutionContext. This method has no effect if the execution context does not currently execute an SQL operation.

- getMaxFieldSize

`public int getMaxFieldSize()`

Returns the maximum number of bytes that are returned for any character column in queries that use the given execution context. A value of 0 means that the maximum number of bytes is unlimited.

- getMaxRows

  `public int getMaxRows()`

  Returns the maximum number of rows that are returned for any query that uses the given execution context. A value of 0 means that the maximum number of rows is unlimited.

- getQueryTimeout

  `public int getQueryTimeout()`

  Returns the maximum number of seconds that SQL operations using this context may take to complete.

- getNextResultSet

  `public ResultSet getNextResultSet() throws SQLException`

  After a stored procedure call, this method returns a result set from the stored procedure. Each call to `getNextResultSet` closes the result set that was retrieved by the previous call. A value of null means that there are no more result sets to be returned.

- getWarnings

  `public SQLWarning getWarnings()`

  Returns the first warning that was reported by the last SQL operation that was executed using this context. Subsequent warnings are chained to the first warning.

- setBatching

  `public void setBatching(boolean batching)`

  Turns batching for that execution context on or off. It does not affect a pending batch, that is, calling this method does not cause an existing batch to be cancelled or executed.

- setBatchLimit

  `public void setBatchLimit(int limit)`

  Causes the batch to automatically execute when *limit* number of statements have been added to the batch, where *limit* is a positive integer or one of `ConnectionContext.UNLIMITED_BATCH` or `ConnectionContext.AUTO_BATCH`. The latter leaves the actual batch size at the discretion of the SQLJ runtime.

  This method does not affect an existing batch.

- executeBatch

  `public int[] executeBatch() throws java.sql.SQLException`

  Executes the pending statement batch contained in this execution context and returns the result as an array of update counts. If no pending statement batch exists for this execution context, null is returned.

- setMaxFieldSize

  `public void setMaxFieldSize(int max)`

  Specifies the maximum number of bytes that are returned for any character column in queries that use the given execution context. The default is 0, which means that the maximum number of bytes is unlimited.

- setMaxRows

  ```
  public void setMaxRows(int max)
  ```

  Specifies the maximum number of rows that are returned for any query that uses the given execution context. The default is 0, which means that the maximum number of rows returned is unlimited.

- setQueryTimeout

  ```
  public void setQueryTimeout(int timeout)
  ```

  Sets the maximum number of seconds that subsequent SQL operations using this context may take to complete.

10.8 Interoperability between JDBC and SQLJ

JDBC and SQLJ are by no means mutually exclusive. In fact, SQLJ is built on top of JDBC. For example, when you use uncustomized serialized profiles, the SQLJ runtime uses JDBC functionality to prepare and execute the SQL statements in the profile dynamically (see “More about profiles” on page 153).

Not only does the SQLJ implementation use JDBC under the covers, it is also perfectly possible to mix SQLJ and JDBC in the same program. In fact, you may have to, because SQLJ has no syntax for embedded dynamic SQL, so if your SQLJ program needs to execute dynamic SQL, you must code those as JDBC calls.

SQLJ provides JDBC interoperability by providing constructs to convert a JDBC result set into an SQLJ iterator and vice versa. This makes it possible, for example, to construct and execute a query dynamically using string operations and JDBC calls, and evaluate the query’s result set using SQLJ syntax.

10.8.1 Converting a JDBC result set into an SQLJ iterator

To convert a JDBC result set into an SQLJ iterator, you use the CAST construct (also known as the iterator conversion statement).

```
#sql iter = { CAST :resultset };
```

Where *iter* is an instance of a public SQLJ iterator class, and `resultset` is a JDBC ResultSet object. Obviously, the number columns in the result set must match the number of columns in the iterator declaration, and the columns must have compatible types. In the case of named iterators, the column names of `resultset` and `iter` must match as well.

10.8.2 Converting an SQLJ iterator into a JDBC result set

You convert an SQLJ iterator into a JDBC result set via the `getResultSet()` method of the iterator class.

```
ResultSet rs = iter.getResultSet();
```

Note that, after converting the iterator to a `ResultSet` object, you may subsequently fetch the rows only by using the ResultSet. Do not use FETCH INTO (in the case of positioned iterators) or the column accessor methods (in the case of named iterators). The results are implementation defined, which is another way of saying that it may or may not work as you expect. (It does work with the DB2 Universal Driver for Java Common Connectivity, but again, it is not guaranteed to work with other SQLJ implementations, and should be avoided.)

11

SQLJ revisited

In this chapter we develop a more realistic SQLJ application, demonstrating various techniques such as:

- Inserting, updating, and deleting rows
- Single-row select (`SELECT INTO`)
- Iterating through result sets (`FETCH`)
- Positioned `UPDATE` and `DELETE`
- Inserting and retrieving LOB data

11.1 Introduction

In the following sections we create a simple SQLJ application that reads and updates one of the sample tables that come with DB2.

We assume that you have the necessary authority to access the sample tables. If this is not the case, please ask your DBA to create a copy of the sample tables and grant you the required authority.

For your convenience, Table 11-1 summarizes the columns in the EMP sample table. However, for the sake of brevity, we will not read or write all of the columns, but only the required (non-null) columns and a few other interesting ones.

Table 11-1 Employee sample table

Column name	SQL data type	nulls
EMPNO	CHAR(6)	No
FIRSTNME	VARCHAR(12)	No
MIDINIT	CHAR(1)	No
LASTNAME	VARCHAR(15)	No
WORKDEPT	CHAR(3)	Yes
PHONENO	CHAR(4)	Yes
HIREDATE	DATE	Yes
JOB	CHAR(8)	Yes
EDLEVEL	SMALLINT	Yes
SEX	CHAR(1)	Yes
BIRTHDATE	DATE	Yes
SALARY	DECIMAL(9,2)	Yes
BONUS	DECIMAL(9,2)	Yes
COMM	DECIMAL(9,2)	Yes

11.2 Creating the Employee class

In this section we develop a Java class that represents the data in the EMP sample table. The class will have attributes that map to the individual columns in the table, and methods to insert, delete, and modify employee data.

To create the SQLJ source file in WSAD, right-click the package then select **New** → **Other** → **Data** → **SQLJ** → **Sqlj File**. Click **Next** and enter the file name, `Employee`, and make sure it is created in the correct package (com.ibm.itso.sg246435.sqlj). (This is similar to what we did in “Creating the source file” on page 142 for our initial SQLJ Hello program.)

In the editor window, we create the skeleton of our Employee class (Example 11-1 on page 201). We add the import statement for java.lang.BigDecimal and the instance variables (also called *fields* or *attributes*) that will hold the individual values from the table. To keep the example reasonably small, we only create fields for a few of the columns.

We also declare a connection context class, an instance of which will be passed to all methods that perform database operations (see “Connection contexts” on page 188 for a detailed explanation of connection contexts).

Example 11-1 Employee.sqlj skeleton

```
package com.ibm.itso.sg246435.sqlj;

import java.math.BigDecimal;

import java.sql.Date;
import java.sql.SQLException;

public class Employee {

  #sql public static context Ctx with (dataSource="jdbc/empdb");            1

  private final String empNo;
  private String firstName;
  private String middleInitial;
  private String lastName;
  private Date hireDate;
  private Short educationLevel;
  private boolean male;
  private BigDecimal salary;
}
```

Note on Example 11-1:

1. This declares an SQLJ connection context. The `with(dataSource…)` declaration causes the SQLJ runtime to create a no-argument constructor, which tries to set up a connection using a JNDI lookup, as explained in “Creating an instance of the context class” on page 190. We use this mechanism in Part 4, “Accessing DB2 from Web applications” on page 247.

Observe that we choose the attribute corresponding to the SEX column in the table to be a boolean variable (rather than a string variable). When inserting data into the table, we will have to make sure that the boolean value “true” is mapped do the string constant “M”, and the boolean value “false” is mapped to “F”. Vice versa, when reading data from the table, we convert “M” to “true” and “F” to “false”.

11.2.1 Implementing attributes, accessors, and constructors

The next step is to create *accessors* for the fields. An accessor is a method to retrieve or modify a field’s value (also called a `getter` or `setter` method, respectively). In Java, it is good practice to always access fields using an accessor method, rather than referring to the field by name.

WSAD can create the accessor methods for you automatically. Right-click anywhere in the editor window, then select **Source -> Generate Getter and Setter**. The dialog that opens allows us to specify which attributes to generate accessors for (Figure 11-1 on page 202).

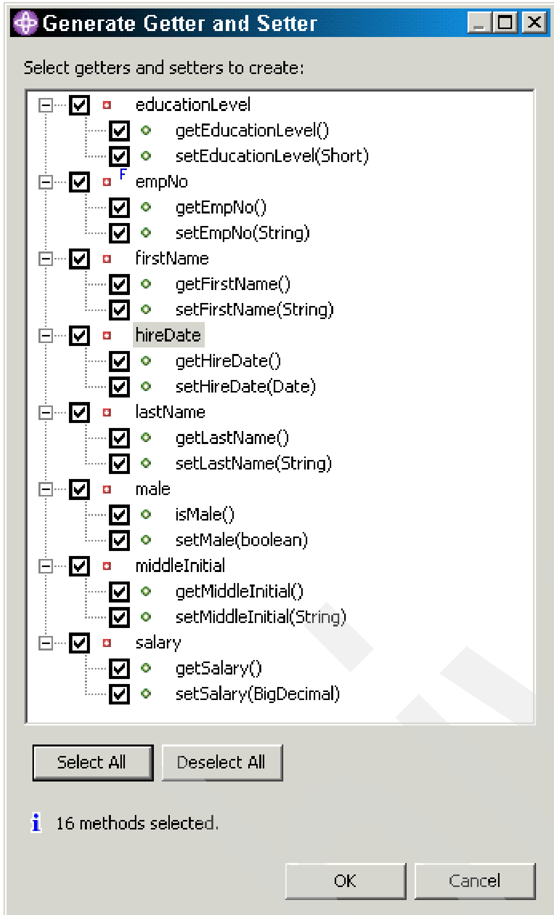

Figure 11-1 Generate getter and setter

We click **Select All** to check all of them, and then **OK**. Since you cannot assign values to a final field, WSAD suggests not to create a setter method for the empNo field (Figure 11-2). We reply **Yes**.

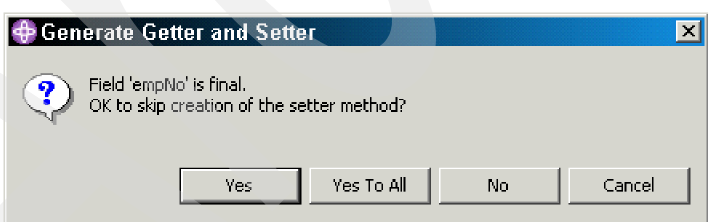

Figure 11-2 Skip setter method

The following code has been added to the Employee.sqlj file (Example 11-2).

Example 11-2 Employee.sqlj with attributes and accessors

```
/**
 * @return
 */
public Short getEducationLevel() {
   return educationLevel;
}
public String getEmpNo() {
```

```
      return empNo;
   }
   public String getFirstName() {
      return firstName;
   }
...
...
   /**
    * @param
    */
   public void setEducationLevel(Short short1) {
      educationLevel = short1;
   }
   public void setFirstName(String string) {
      firstName = string;
   }
...
...
```

Note that there is no setter method for empNo, since we told WSAD not to generate one.

Now a red marker appears on the left-hand side of the editor window, indicating an error. The error message says that `the blank final variable empNo may not have been initialized`. This is because a variable that is declared final must be assigned an initial value somewhere, either in the variable declaration itself or in the constructor.

11.2.2 Implementing the constructor to create new employees

Finally, we have to create a constructor to create new employees. The constructor has one argument, the employee number. Remember that we did not create a setter method for the employee number, so the only way to assign it is via the constructor.

Example 11-3 Employee constructor

```
public Employee(String empNo) {
  this.empNo = empNo;
}
```

Insert the constructor somewhere in the class body. The preferred Java convention is to put constructors right between the field declarations and the other methods.

After saving the file, the error message about the missing initialization of empNo will have gone away, since now there exists a constructor that initializes it.

11.2.3 Implementing the `insert()` method

Now we are ready to code the individual methods for inserting new employees into the table, for deleting employees, and for updating employee data.

We begin with the `insert()` method, which creates a new employee. The most common form of an INSERT statement looks like this:

```
INSERT INTO table_name (column_list) VALUES (value_list)
```

Where:

- *table_name* is the name of the table (or view) into which you want to insert.
- *column_list* is a comma-separated list of column names from that table.

- *value_list* is a comma-separated list of values to be inserted into the respective columns.

The values supplied in the *value_list*, in turn, fall into two very different categories. They can be:

- *SQL expressions*, that is, expressions defined by the SQL language.
- *host variables*, that is, references to a variable defined in the host language (in our case, Java variables, which can in turn be local variables, method parameters, or fields).

A sample SQLJ insert statement is shown in Example 11-4.

Example 11-4 The Employee.insert() method

```
public void insert(Ctx ctx) throws SQLException {                              1, 2
  #sql [ctx] {                                                                  3
    INSERT INTO DSN8710.EMP (
        EMPNO                                                                   4
      , FIRSTNME
      , MIDINIT
      , LASTNAME
      , HIREDATE
      , SEX
      , SALARY
    ) VALUES (
        :empNo                                                                  5
      , :firstName
      , :middleInitial
      , :lastName
      , :hireDate
      , :(male ? "M" : "F")                                                     6
      , :salary
   )
  };
}
```

Notes on Example 11-4:

1. The `insert()` method takes an instance of the context class as a parameter. We could also have stored that instance in a static variable of the class, but this is not considered good practice.
2. Since executing the SQLJ statement may fail, we have to declare that the method may throw an `SQLException`.
3. Use `ctx` as the connection context for the `INSERT` operation.
4. The list of columns to be inserted. Note that, for brevity, we do not supply values for all of the columns in the EMP sample table.
5. The list of values for the corresponding columns. The number of values must match the number of columns to be inserted.
6. We make use of the Java conditional operator to convert the boolean value of the `male` attribute to the corresponding string value. Note that expressions (as opposed to host variables) must be parenthesized.

11.2.4 Creating a test driver

At this point, we are ready to run a first test and verify that the `insert()` method works.

Rather than implementing the test code in the Employee class itself, we create a small test class, which creates a new Employee object and calls `insert()` to create the record in the database. We call the class `EmployeeTest` and put it in the same Java package. Since the program will not contain SQL statements itself, we can create a regular Java class (as opposed to an SQLJ source file). The easiest way to create the skeleton source code for the class is to use the New Class wizard (see “Creating the project” on page 102).

Example 11-5 shows the source code of the `EmployeeTest` class.

Example 11-5 The EmployeeTest class

```
package com.ibm.itso.sg246435.sqlj;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.math.BigDecimal;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class EmployeeTest {

  private static Employee.Ctx ctx;                                       1

  /**
   * Load JDBC driver and initialize SQLJ connection context.
   *
   * Driver name, URL and driver-specific properties
   * are read from a configuration file.
   */
  private static void initialize()                                       2
    throws ClassNotFoundException, SQLException, FileNotFoundException, IOException {

    // Read the properties file
    FileInputStream propsFile = new FileInputStream("EmployeeTest.properties");
    Properties properties = new Properties();
    properties.load(propsFile);
    propsFile.close();

    // Get driver property, and load the driver
    String driver =                                                      3
      properties.getProperty("driver", "com.ibm.db2.jcc.DB2Driver");
    Class.forName(driver);                                               4

    String url = properties.getProperty("url");                          5
    Connection conn = DriverManager.getConnection(url, properties);

    // Set up the connection context with autocommit disabled
    ctx = new Employee.Ctx(conn);                                        6
  }

  public static void testInsert() throws SQLException {
    Employee emp = new Employee("000042");                               7
    emp.setFirstName("Arthur");
    emp.setLastName("Dent");
    emp.setMale(true);
    emp.setSalary(new BigDecimal(200000)); // Wish it were so...
    emp.insert(ctx);                                                     8
```

```
    System.out.println("Employee " + emp + " created successfully.");
  }

   public static void main(String[] args) {
      try {
         initialize();
         testInsert();
      } catch (SQLException e) {
         System.err.println(e +". SQLCODE = " + e.getErrorCode());
      } catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

Notes on Example 11-5 on page 205:

1. Declare an instance of the connection context class. For simplicity, we decided to store that instance in a class variable so it need not be passed to each test method as a parameter, as we do in the Employee class.
2. This method initializes the SQLJ connection context. Other than in previous examples, the connection properties are not hardcoded; rather, they are read in from a file. (Note that this technique only works for String-valued properties.)
3. Get the driver name from the properties; if not present, use the JCC driver as a default.
4. Load the JDBC driver.
5. Get the URL property, and open the connection.
6. Create an SQLJ connection context using that connection.
7. First, we create and initialize an Employee instance.
8. Then, we call the Employee.insert() method, passing it the connection context.

As explained above, this program does not use hardcoded values for the JDBC driver name, URL, username and password; rather it expects them in a properties file. Create the properties file directly in the SG24-6435 folder (right-click the folder name and select **New** → **File**. Enter `EmployeeTest.properties` for the file name (this file name is hardcoded in the program). Type your connection properties (Example 11-6); lines beginning with a # sign are comment lines.

Example 11-6 Sample properties file for the EmployeeTest program

```
# Class name of the JDBC driver
driver = com.ibm.db2.jcc.DB2Driver

# URL, username, and password
url =jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y:retrieveMessagesFromServerOnGetMessage=TRUE;
user = bartr3
password = secret
```

Now try and run the test program. You should normally receive the following message:

```
com.ibm.db2.jcc.c.SQLException: AN UPDATE, INSERT, OR SET VALUE IS NULL, BUT THE OBJECT
COLUMN  CANNOT CONTAIN NULL VALUES. SQLCODE = -407
```

This is because we did not supply a value for a NOT NULL column in the table (see Table 11-1 on page 200); the middle initial column must be specified.

Of course, it would be easy to change the test program by adding:

```
emp.setMiddleInitial(“ “);
```

But a better approach is to modify the Employee class and ensure that the middle initial is always initialized to an appropriate default value (a string containing just one blank).

Find the declaration of the middle initial field in Employee.sqlj and modify it to initialize the field with a default value:

```
private String middleInitial = " ";
```

While we are at it, we implement a `toString()` method to return a printable representation of an Employee object (Example 11-7).

Example 11-7 Create printable representation

```
/**
  * Return a printable representation
  */
public String toString() {
   return getLastName() + ", " + getFirstName() + " " + getMiddleInitial();
}
```

Retranslate Employee.sqlj, then run EmployeeTest again. This time, the program should print:

```
Employee Dent, Arthur   created successfully.
```

Without the `toString()` method, the program would have printed something like `Employee com.ibm.itso.sg246435.sqlj.Employee@420f0a00 created successfully`.

Run EmployeeTest one more time. This time it should print:

```
com.ibm.db2.jcc.c.SqlException: AN INSERTED OR UPDATED VALUE IS INVALID BECAUSE INDEX IN
INDEX SPACE XEMP1 CONSTRAINS COLUMNS OF THE TABLE SO NO TWO ROWS CAN CONTAIN DUPLICATE
VALUES IN THOSE COLUMNS. RID OF EXISTING ROW IS X'0000001221'. SQLCODE = -803
```

Since we violated a unique constraint. We tried to insert another record with the same primary key value.

11.2.5 Verifying that the program worked

After running the test driver, a new row should have been added to the DSN8710.EMP table. If you are familiar with QMF or SPUFI, you could of course now go ahead and use one of these to verify it. We suggest, however, that you use the WSAD Data Perspective (if only to get a feel of it).

“Using the data perspective” on page 74 explains how to set up a database connection for the data perspective and how to display the sample contents of a DB2 table.

Select the DSN8710.EMP table, then select **Sample Contents…** from the context menu. The result is shown in Figure 11-3 on page 208.

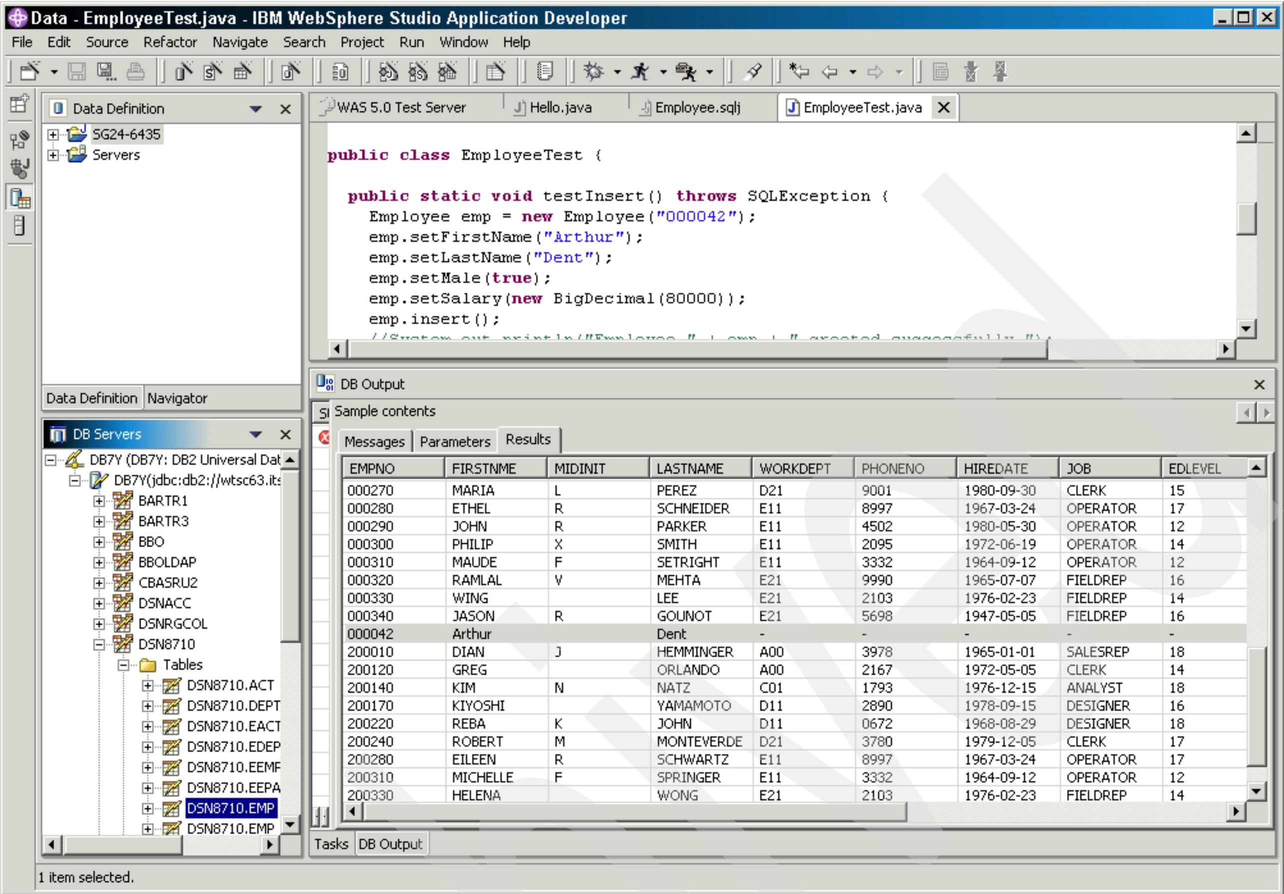

Figure 11-3 Sample contents with Dent inserted

11.2.6 Implementing the findByPrimaryKey() method

Now that we are able to create a new employee, we need a method to find an existing employee, given his or her employee number.

Since we do not have an object to work with—after all, we want to find one in the first place—`findByPrimaryKey()` cannot be an instance method; it has to be a class method (in Java terms, it must be declared static).

We implement this method using the `SELECT INTO` SQL statement. You use `SELECT INTO` whenever you expect exactly one result row as the result from a query (singleton select). When DB2 detects that no row or more than one row satisfies the `WHERE` condition, an `SQLException` will be raised. (The latter case cannot occur in our example since `EMPNO`, being the primary key column, does not allow duplicate values.)

We have a little ‘problem’ with the SEX column, since we decided to map it to a boolean variable. Therefore, we have to convert the string value in the column to a boolean. In Example 11-8 below, we do this by SELECTing the value into an intermediate, local String variable, then inspecting that variable’s contents, and setting the attribute’s value accordingly.

Example 11-8 Employee.findByPrimaryKey (first version)

```
public static Employee findByPrimaryKey(Ctx ctx, String empNo) throws SQLException {
  Employee emp = new Employee(empNo);
  String sex = null;                                                    1
```

```
  #sql [ctx] {
    SELECT FIRSTNME
         , MIDINIT
         , LASTNAME
         , HIREDATE
         , SEX
         , SALARY
      INTO :(emp.firstName)
         , :(emp.middleInitial)
         , :(emp.lastName)
         , :(emp.hireDate)
         , :sex                                                    2
         , :(emp.salary)
       FROM DSN8710.EMP
      WHERE EMPNO = :empNo
  };
  if (sex.equals("M"))                                             3
    emp.male = true;
  else
    emp.male = false;
  return emp;
}
```

Notes on Example 11-8 on page 208:

1. This declares the local variable to receive the value of the SEX column.
2. Here we SELECT INTO the local variable instead of an Employee instance variable.
3. Inspect the local variable's value and set the value of the male attribute accordingly.

However, there is a more elegant solution. There is a built-in conversion in SQLJ from numeric data types to boolean: The value 0 (zero) maps to false, whereas any non-zero value maps to true. So, we need an SQL expression that converts the string 'F' to zero and any other string value to non-zero. The following CASE expression does the trick:

```
CASE SEX WHEN 'M' THEN 1 ELSE 0 END
```

Example 11-9 shows the updated source code. Note that we can now specify the instance variable directly as a host variable in the SQL statement.

Example 11-9 Employee.findByPrimaryKey (second version)

```
public static Employee findByPrimaryKey(Ctx ctx, String empNo) throws SQLException {
  Employee emp = new Employee(empNo);
  #sql [ctx] {
    SELECT FIRSTNME
         , MIDINIT
         , LASTNAME
         , HIREDATE
         , CASE SEX WHEN 'M' THEN 1 ELSE 0 END
         , SALARY
      INTO :(emp.firstName)
         , :(emp.middleInitial)
         , :(emp.lastName)
         , :(emp.hireDate)
         , :(emp.male)
         , :(emp.salary)
       FROM DSN8710.EMP
      WHERE EMPNO = :empNo
  };
```

```
  return emp;
}
```

To test the `findByPrimaryKey()` method, we insert a call to the `Employee.findByPrimaryKey()` method, and we comment out the call to `testInsert()` for the time being—otherwise, we would not be able to run the test program due to the unique key constraint violation we saw in “Creating a test driver” on page 204.

Example 11-10 shows the updated `main()` method.

Example 11-10 Modified EmployeeTest.main()

```
public static void main(String[] args) {
  try {
    initialize();
    // testInsert();
    Employee emp = Employee.findByPrimaryKey(ctx,"000042");
    System.out.println("Successfully retrieved " + emp);
      } catch (SQLException e) {
         System.err.println(e +". SQLCODE = " + e.getErrorCode());
      } catch (Exception e) {
         System.err.println(e);
      }
}
```

Before running the test program again, do not forget to re-translate Employee.sqlj. The test program should print:

```
Successfully retrieved Dent, Arthur
```

Before we can un-comment the call to `testInsert()` again, we have to write a method to delete employee records, which we do in the following section.

11.2.7 Implementing the delete() method

The delete() method deletes an Employee by running an SQL DELETE statement against the database. Example 11-11 shows the source code.

Example 11-11 Employee.delete() method

```
public void delete(Ctx ctx) throws SQLException {
  #sql [ctx] {
    DELETE FROM DSN8710.EMP
     WHERE EMPNO = :empNo
  };
}
```

In our TestEmployee class, we add a call to delete() to the `main()` method. We also verify that `delete()` worked, by trying to retrieve the employee record again. This should raise an `SQLException`, which we catch.

Example 11-12 shows the updated `main()` method.

Example 11-12 EmployeeTest.main() after adding the call to delete()

```
public static void main(String[] args) {
  try {
    initialize();
    // testInsert();
```

```
    Employee emp = Employee.findByPrimaryKey("000042");
    System.out.println("Successfully retrieved " + emp);
    emp.delete(ctx);
    try {
      emp = Employee.findByPrimaryKey(ctx, "000042");
    } catch (SQLException expected) {
      if (expected.getSQLState().equals("02000")) {
        System.out.println("Employee record deleted successfully");
      } else {
        // Different SQL exception -- we didn't really expect it
        throw expected;
      }
    }

      } catch (SQLException e) {
         System.err.println(e +". SQLCODE = " + e.getErrorCode());
      } catch (Exception e) {
         System.err.println(e);
      }

}
```

11.2.8 Implementing the update() method

The `update()` method writes back any changes made to an Employee object. In the WHERE clause of the UPDATE statement, we specify that we want to update the single row identified by the table's primary key (the employee number). Other than that, the `update()` method looks pretty similar to the `insert()` method; again, we use the conditional operator to convert the boolean attribute to an 'M' or 'F'.

Example 11-13 Employee.update() method

```
public void update(Ctx ctx) throws SQLException {
  #sql [ctx] {
    UPDATE DSN8710.EMP
       SET FIRSTNME = :firstName
         , LASTNAME = :lastName
         , MIDINIT  = :middleInitial
         , HIREDATE = :hireDate
         , SEX      = :(male ? "M" : "F")
         , SALARY   = :salary
     WHERE EMPNO    = :empNo
  };
}
```

If you, for example, want to change the last name (after retrieving the employee), you can add the following code (Figure 11-14) to your EmployeeTest program.

Example 11-14 Invoking the Employee.update() method

```
Employee emp = Employee.findByPrimaryKey("000042");
System.out.println("Successfully retrieved " + emp);
emp.setLastName("Dentist");
emp.update(ctx);
System.out.println("Successfully updated " + emp);
```

11.2.9 Implementing the findAll() method

Next we need a method to retrieve all employee records from the table. Again, since we do not necessarily have an Employee object to work with, it is going to be a static method of the Employee class (Example 11-15).

Because we retrieve multiple records, we need a way to iterate through them, one at a time. The findAll() method therefore returns an iterator, which the caller can pass to the `fetch()` method described later. It is the caller's responsibility to properly close the iterator when done with it.

Example 11-15 Employee.findAll() method

```
public static EmployeeIterator findAll(Ctx ctx) throws SQLException {
  EmployeeIterator iter;
  #sql [ctx] iter = {
    SELECT EMPNO
          ,FIRSTNME
         , MIDINIT
         , LASTNAME
         , HIREDATE
         , CASE SEX WHEN 'M' THEN 1 ELSE 0 END
         , SALARY
      FROM DSN8710.EMP
  };
  return iter;
}
```

Next, we declare the iterator class. The iterator columns must correspond in number and data type to the columns in the SELECT list (see "Iterator declarations" on page 177 for a detailed explanation of iterator declarations). The iterator declaration can go anywhere in the class body, for example, right before the `findAll()` method.

Example 11-16 Declaration of EmployeeIterator

```
#sql public static iterator EmployeeIterator (
    String        // EMPNO
  , String        // FIRSTNME
  , String        // MIDINIT
  , String        // LASTNAME
  , Date          // HIREDATE
  , boolean       // SEX
  , BigDecimal    // SALARY
);
```

Tip: Note that WSAD will flag the statement when you code the `findAll()` method for the EmployeeIterator class (EmployeeIterator cannot be resolved). You can happily ignore this message. This message is displayed because WSAD does not understand that an SQLJ iterator declaration results in a class file until after translation. To put this to the test, save the SQLJ file in WSAD. This will invoke the SQLJ translator, and the Java compiler. If everything is fine, you should not receive any errors. (In WSAD 5.1.1 the use of a iterator class in a method return will no longer be flagged as an error.)

Finally, we need a method to fetch the current row and construct an Employee object (Example 11-17 on page 213). Again, this method is declared static since we do not have an object to work with.

Unlike the other methods, the `fetch()` method need not be passed a connection context. This is analogous to the situation in JDBC where you do not need a `Connection` object in order to work with a `ResultSet`. Instead, we pass an iterator instance (that had been returned by a previous call to `findAll()`).

Example 11-17 Employee.fetch() method

```
public static Employee fetch(EmployeeIterator iter) throws SQLException {
  String empno = null;                                                    1
  String firstName = null;
  String middleInitial = null;
  String lastName = null;
  Date hireDate = null;
  boolean male = false;
  BigDecimal salary = null;
  #sql {
   FETCH :iter                                                            2
    INTO :empno
       , :firstName
       , :middleInitial
       , :lastName
       , :hireDate
       , :male
       , :salary
  };
  if (iter.endFetch())                                                    3
    return null;
  Employee emp = new Employee(empno);                                     4
  emp.firstName = firstName;
  emp.lastName = lastName;
  emp.middleInitial = middleInitial;
  emp.hireDate = hireDate;
  emp.male = male;
  emp.salary = salary;
  return emp;
}
```

Notes on Example 11-17:

1. Declare host variables into which the current row will be fetched. For technical reasons, we have to initialize all host variables; otherwise, the Java compiler will complain about the use of a possibly unassigned variable (although we do know that they have been assigned if the FETCH statement was successful).
2. Tries to fetch the current row into the host variables declared in note 1 that, as explained above, we do not need to specify a connection context (in fact, the SQLJ translator issues a warning if you do).
3. Return `null` if no more rows were found.
4. Otherwise, create and populate an `Employee` object, and return it.

Now it is time to test our `findAll()` method. First we code a `testFindAll()` method in our testEmployee program (Figure 11-18).

Example 11-18 testFindAll() method

```
public static void testFindAll() throws SQLException {
  EmployeeIterator iter = null;
  try {
    iter = Employee.findAll(ctx);
    for (;;) {
```

```
      Employee emp = Employee.fetch(iter);
      if (emp == null) break;
      System.out.println(emp);
    }
  } finally {
    if (iter != null) iter.close();
  }
}
```

Do not forget to add an import statement for the EmployeeIterator in the beginning of your EmployeeTest.java program, like:

```
import com.ibm.itso.sg246435.sqlj.Employee.EmployeeIterator;
```

Then we invoke it from within the main program (Figure 11-19).

Example 11-19 Invoking the testFindAll() method

```
public static void main(String[] args) {
  try {
    initialize();
    ...
    testFindAll();
    ...
  } catch (Exception e) {
    ...
  }
}
```

11.2.10 Working with LOB data: The getPicture() and setPicture() methods

To add a finishing touch to our program, wouldn't it be nice if we could retrieve and create an employee's picture? We do that in this section, introducing the data types and methods for working with LOB (Large OBject) data.

LOBs come in two flavors:

- Character Large OBjects (CLOBs) for holding character data, such as XML documents
- Binary Large OBjects (BLOBs) for holding binary data, such as multimedia files

Working with LOB data is almost as easy in Java as working with simple data types, as you will see in the following examples.

Retrieving employee photos

Example 11-20 shows the first version of the `getPicture()` method. The `SELECT` statement looks exactly the same as with non-LOB columns. The LOB data is returned as a byte array, which represents the employee's picture in GIF format.

Example 11-20 Employee.getPicture(), first version

```
/**
 * Retrieves the employee's picture.
 *
 * @return
 *   An InputStream to read the picture in GIF format,
 *   or <code>null</code> if no picture is available.
 *
 * @exception SQLException
 *   A database error occurred.
```

```
 */
public byte[] getPicture(Ctx ctx) throws SQLException {
  byte[] picture;
  #sql [ctx] {
    SELECT BMP_PHOTO
      INTO :picture
      FROM DSN8710.EMP_PHOTO_RESUME
     WHERE EMPNO = :empNo
  };
  return picture;
}
```

However, there is one problem with the approach in Example 11-20 on page 214. When you retrieve the data into a byte array, the SQLJ runtime has to read the entire LOB in one big chunk. Obviously, this can cause problems when working with very large LOBs, such as video clips. For this reason, Java offers two interfaces to work with LOB data, namely, java.sql.Blob for BLOB data, and java.sql.Clob for CLOB data. Both offer similar functionality and differ only in that Blob treats the data as binary, whereas Clob treats it as text.

java.sql.Blob

The java.sql.Blob interface offers the following methods:

- `long length() throws SQLException`

  Returns the number of bytes in the BLOB
- `byte[] getBytes(long pos, int length) throws SQLException`

  Reads up to `length` bytes from the BLOB, starting at position `pos`
- `java.io.InputStream getBinaryStream() throws SQLException`

  Retrieves the BLOB as a stream
- `long position(byte pattern[], long start) throws SQLException`
  `long position(Blob pattern, long start) throws SQLException`

  Searches the BLOB for a byte pattern

java.sql.Clob

The java.sql.Clob interface offers the following methods:

- `long length() throws SQLException`

  Returns the number of characters in the CLOB
- `String getSubString(long pos, int length) throws SQLException`

  Get a part of the CLOB value, starting at position pos and up to length characters long
- `java.io.Reader getCharacterStream() throws SQLException`

  Retrieves the CLOB as a Unicode stream
- `java.io.InputStream getAsciiStream() throws SQLException`

  Retrieves the CLOB as a stream of ASCII bytes
- `long position(String searchstr, long start) throws SQLException`
  `long position(Clob searchstr, long start) throws SQLException`

  Searches for an occurrence of a String in the CLOB

Again, it is important to point out that a Blob or Clob object does not actually hold the entire LOB data; rather, it offers methods to retrieve chunks of the LOB object, or to retrieve a stream from which to read the LOB data in manageable pieces.

Example 11-21 shows an updated version of the `getPicture()` method, which uses a Lob object, returning an `InputStream`, which can then be used by the caller to retrieve the LOB data.

Also, the updated version checks whether a picture has actually been found. If not, this is not considered an error, so an `SQLException` will not be thrown. The applications return a null value instead.

Example 11-21 Employee.getPicture(), second version

```
import java.io.InputStream;
...
/**
 * Retrieves the employee's picture.
 *
 * @return
 *   An InputStream to read the picture in GIF format,
 *   or <code>null</code> if no picture is available.
 *
 * @exception SQLException
 *   A database error occurred.
 */
public InputStream getPicture(Ctx ctx) throws SQLException {
  Blob picture;
  try {
    #sql [ctx] {
      SELECT BMP_PHOTO
        INTO :picture
        FROM DSN8710.EMP_PHOTO_RESUME
       WHERE EMPNO = :empNo
    };
  } catch (SQLException sqlException) {
    if (sqlException.getSQLState().equals("02000")) // row not found
      return null;
    else
      throw sqlException;
  }
  return picture.getBinaryStream();
}
```

Again, we test the method by creating and calling a test method in our EmployeeTest class. The test method, shown in Example 11-22, expects an employee number and a directory where to save the picture file.

Example 11-22 EmployeeTest.testGetPicture()

```
import java.io.File;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileOutputStream;
...
public static void testGetPicture(String empno, File dir) throws SQLException,
                                                                  IOException {
  Employee emp = Employee.findByPrimaryKey(ctx,empno);
  InputStream in = null;
  OutputStream out = null;
  try {
    in = emp.getPicture(ctx);
    if (in == null) {
      System.out.println("No picture available for employee " + empno);
```

```
    } else {
      out = new FileOutputStream(new File(dir, "Emp" + empno + ".bmp"));
      int nread;
      byte[] buf = new byte[1024];
      while ((nread = in.read(buf)) > 0)
        out.write(buf, 0, nread);
      System.out.println("Picture retrieved for employee " + empno + " in "
                         + dir + " with name Emp" + empno + ".bmp");
    }
  } finally {
    if (in != null) in.close();
    if (out != null) out.close();
  }
}
```

The DB2 sample database contains several employees for whom a picture is available, including employee #000130. We modify the main method of EmployeeTest, by adding:

```
testGetPicture("000130", new File("C:/temp"));
```

After running EmployeeTest once more, a file named Emp000130.bmp is created in the C:\temp directory, which you can view with a Web browser.

Storing employee photos in the database

Now that we are able to retrieve employee photos, we also want to store them in the database.

Example 11-23 shows the `createPicture()` method, which takes an `InputStream` argument from which it reads the picture in BMP format, and a length argument that gives the size of the bitmap, in bytes.

Also note that we called the method `createPicture()` rather than `setPicture()`, since the method only allows creating a picture, not replacing an existing one. If we want that functionality, we first would have to figure out if a picture already exists, then perform either an INSERT or an UPDATE statement. Implementing this is left as an exercise.

Example 11-23 Employee.createPicture()

```
import sqlj.runtime.BinaryStream;
import java.io.IOException;
import java.io.File;
import java.io.FileInputStream;
...

/**
 * Creates the employee's picture.
 *
 * @param picture
 *   An InputStream supplying the picture in BMP format.
 *   On return from this method, the stream is closed.
 *
 * @exception SQLException
 *   A database error occurred.
 * @exception IOException
 *   An error occurred reading the input stream.
 */
public void createPicture(Ctx ctx, InputStream picture, int length)         1
  throws SQLException, IOException
{
```

```
  try {
    BinaryStream lobStream = new BinaryStream(picture, length);             2
    #sql [ctx] {
      INSERT INTO DSN8710.EMP_PHOTO_RESUME (
      EMPNO
    , BMP_PHOTO
      ) VALUES (
      :empNo
    , :lobStream
      )
    };
  } finally {
    picture.close();
  }
}
```

Notes on Example 11-23 on page 217:

1. Since we cannot know the picture's size (other than by reading the input stream untill the end), it must be passed in as a parameter by the caller.
2. The preferred host variable type for BLOBs is BinaryStream. The SQLJ runtime wants to know the BLOB's size beforehand, so we have to pass it in with the constructor for BinaryStream.

   Alternatively, we could have read the entire picture into a byte array ourselves, and pass the byte array as host variable.

As a convenience, we implement another `createPicture()` method (Example 11-24), which takes a file name argument, then determines the length of the file and calls the original `createPicture()` method. We add that one to the Employee class as well.

Example 11-24 Overloaded createPicture() method

```
/**
 * Creates the employee's picture.
 *
 * @param pictureFilename
 *   Name of a file containing the picture in BMP format.
 *
 * @exception SQLException
 *   A database error occurred.
 * @exception IOException
 *   An error occurred reading the file.
 */
public void createPicture(Ctx ctx, String pictureFilename) throws SQLException, IOException
{
  File pictureFile = new File(pictureFilename);
  createPicture(ctx, new FileInputStream(pictureFile), (int) pictureFile.length());
}
```

For testing purposes, we added the `createPicture()` method to the EmployeeTest class (Example 11-25). The only difficult thing is to find a picture of Arthur Dent.

Example 11-25 EmployeeTest.testCreatePicture()

```
Employee emp= Employee.findByPrimaryKey(ctx,"000042");
emp.createPicture(ctx,"C:/temp/arthur_dent.bmp");
System.out.println("Successfully stored picture of " + emp);
```

Important: Up to this point we have not customized the profile and run it as static SQL against the database. To get the most out of SQLJ, it is very important to do this. When using WSAD, you simply have to generate and run your SQLJ Ant script again, as described in “Preparing SQLJ programs to use static SQL through WSAD” on page 157.

12

The DB2 Universal Driver

This chapter discusses some of the enhancements of the DB2 Universal Driver for SQLJ and JDBC, which is a significant reengineering of Java support for DB2, and will eventually replace the existing platform-specific drivers.

In particular, we discuss the following:

- What the DB2 Universal Driver for SQLJ and JDBC is
- Setting connection properties in the URL
- Functionality enhancements

12.1 What the DB2 Universal Driver for SQLJ and JDBC is

The DB2 Universal Driver for SQLJ and JDBC is a significant reengineering of Java support for DB2.

The DB2 Universal Driver for SQLJ and JDBC has the following advantages over the previous solutions:

- One single driver for the Unix, Windows, and z/OS platforms, improving portability and consistent behavior of applications.
- Improved integration with DB2.
- Type 4 driver functionality for thin clients, as well as Type 2 functionality.
- Easier installation and deployment. As a Type 4 driver is completely written in Java, it is easy to deploy; you only have to ship the jar files containing the driver code to the required machine, change the classpath, and you are done. In addition, the use of a Type 4 driver does not require any setup in DB2, such as cataloging entries in the Database, Node, and DCS directories.
- 100 percent Java application development process for SQLJ, as demonstrated in "Preparing an application to use static SQL" on page 157.
- 100 percent JDBC V3 compliance. Since DB2 UDB V8.1.3 (FixPak 3), the Universal Driver is JDBC 3.0 compliant. The version that will ship on z/OS will be JDBC 3.0 compliant from day one.
- Significant performance improvements for both JDBC and SQLJ applications on the Linux, Unix, and Windows platforms.
- Trace improvements.

A architectural overview, as well as some configuration on how to use the new Universal Driver, can be found in "The IBM DB2 Universal Driver for SQLJ and JDBC" on page 27. In this chapter we look in a bit more detail at some of the enhancements that come with the DB2 Universal Driver for SQLJ and JDBC.

12.2 Setting connection properties in the URL

Recall from "Connecting to a database" on page 36 that a JDBC URL has the form:

```
jdbc:driverselection:database-spec
```

Where *driverselection* is db2 for the JCC driver, and *database-spec* is of the form *location-name* (for Type 2 connections), or *hostname:portnumber/instance-name* for Type 4 connections. This was not the whole truth. The IBM DB2 Univerasl Driver for SQLJ and JDBC allows you to specify additional connection properties after the instance-name part or location-name part, separated by colons and taking the form *propertyKey=value*. For the list of supported property keys, see Table 12-1 on page 223.

These properties can be extremely useful, especially for debugging and tracing purposes, as discussed in "Tracing" on page 244.

Table 12-1 Property keys for the DB2 Universal Driver for SQLJ and JDBC

Property key	Description
driverType	Determines the JDBC connectivity type to a data source. If driverType is not set, Type 2 connectivity is selected by default. This property is a data source property, and not a connection property. JDBC 1 connectivity selection is based on the URL syntax.
user	Specifies the user ID when connecting using the DriverManager.getConnection(String url, Properties properties) method.
password	Specifies the password when connecting using the DriverManager.getConnection(String url, Properties properties) method.
serverName	See “Improved security for DB2 authentication” on page 226.
portNumber	The TCP/IP port number where the DRDA server listens for connection requests to this data source. The default value is 446.
databaseName	► For the Type 4 driver, the actual database name, and not the locally catalogued database name. The Universal JDBC Type 4 driver does not rely on information cataloged in the DB2 database directory. ► For the Type 2 driver, this can be the actual database name (if serverName and portNumber are specified), or the locally catalogued database name. The Type 2 driver can connect directly using TCP/IP, or rely on information cataloged in the DB2 database directory. (This applies to LUW.)
logWriter	See “Tracing” on page 244.
traceLevel	See “Tracing” on page 244.
traceFile	See “Tracing” on page 244.
traceFileAppend	If set to true, causes the trace to be appended to the trace file; otherwise, the file is overwritten. See “Tracing” on page 244.
fullyMaterializeLobData	Determines whether the LOB data flowing from the server to the client is blasted (true), or streamed (false).
resultSetHoldability	► For DB2 targets, the default is HOLD_CURSORS_OVER_COMMIT. ► For Cloudscape Network Server, the default is CLOSE_CURSORS_AT_COMMIT.
currentPackageSet	This property is used in conjunction with the db2jdbcbind -collection option, which is given when the JDBC/CLI package is bound during DB2 installation.

Property key	Description
securityMechanism	See “Improved security for DB2 authentication” on page 226.
kerberosServerPrincipal	See “Improved security for DB2 authentication” on page 226.
gssCredential	See “Improved security for DB2 authentication” on page 226.
readOnly	Creates a read-only connection. By default this is false. This property is flowed at connect time for Type 4 connectivity.
deferPrepares	When enabled, server prepare requests are deferred until execute time (reducing network traffic).
currentSchema	Sets the CURRENT SCHEMA special register of the DB2 server. Currently not supported by DB2 Universal Database for OS/390 and z/OS, and will result in either an error or an exception when targeting DB2 Universal Database for OS/390 and z/OS servers.
currentSQLID	Sets the CURRENT SQLID special register on DB2 Universal Database for OS/390 and z/OS. Setting this property will fail when targeting DB2 Universal Database on Windows or UNIX-based platforms.
cliSchema	Indicates the schema of the DB2 shadow catalog tables or views to search when you issue a database metadata catalog query.
retrieveMessagesFromServerOnGetMessage	Enables this property to direct all calls to the standard JDBC SQLException.getMessage() method to invoke a server-side stored procedure that retrieves the formatted message text for the error (the invocation of the stored procedure starts a separate unit of work). By default, this property is disabled and the full message text is *not* returned to the client when a server-side error occurs. The proprietary method DB2Sqlca.getMessage() can also be called to retrieve the formatted message text.
clientUser	The clientUser property establishes the current client user name for this connection. The current client user name is not the JDBC connection user name, but is a string for accounting purposes. Unlike the JDBC connection user name, the current client user name, and any associated client information, may change during the life of this connection. See “Java API for Set Client Information (SQLESETI)” on page 226.
clientWorkstation	Establishes the workstation name for the current client on this connection.
clientApplicationInformation	Establishes generic application information for the current client on this connection.

Property key	Description
clientAccountingInformation	Establishes generic accounting information for the current client on this connection.

You can supply values for the individual keys by either using the DriverManager.getConnection(String url, Properties properties) method (Example 12-1), or by encoding them in the URL (for example, in a DataSource definition, or on the application's command line).

Example 12-1 Setting properties using DriverManager.getConnection(String, Properties)

```
import java.net.InetAddress;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;

...

private static Connection openConnection() {
  Properties properties = new Properties();
  properties.put("user", "bartr1");
  properties.put("password", "secret");
  properties.put("clientWorkstation", InetAddress.getLocalHost().getHostName());
  properties.put("clientUser", System.getProperty("user.name"));

  Connection conn = DriverManager.getConnection(url, properties);
  return con;
}
```

12.3 Functionality enhancements

The DB2 Universal Driver for SQLJ and JDBC provides more functionality than the older drivers, for example, it supports scrollable cursors, batch updates, and savepoints, as discussed in the following sections.

12.3.1 Scrollable cursor support

Scrollable cursor support refers to the possibility of moving a result set's cursor backward as well as forward. This is especially useful for GUI applications, when users want to browse a result set backward and forward. Without scrollable cursors, the application had to either cache the result set in memory, or submit the query again when the user scrolled backward.

This feature, introduced in JDBC 2.0, is now supported in the DB2 Universal Driver for SQLJ and JDBC.

12.3.2 Batch updates

A batch update is a set of multiple update statements that is submitted to the database for processing as a batch. Sending multiple update statements to the database together as a unit can, in some situations, be much more efficient than sending each update statement separately. This ability to send updates as a unit, referred to as the batch update facility, is one of the features provided with the JDBC 2.0 API, and is now supported with the DB2 Universal Driver for SQLJ and JDBC.

Example 12-2 (from the JDBC 3.0 specification) demonstrates how to use batch updates. Note that autocommit should be turned off when using batch updates. Also, each of the statements in the batch must be one that returns an update count (for example, a SELECT statement in the batch is not allowed, and will cause an SQLException to be thrown).

Example 12-2 Creating and executing batched INSERT statements

```
// Turn off autocommit
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
stmt.addBatch("INSERT INTO employees VALUES (1000, 'Joe Jones')");
stmt.addBatch("INSERT INTO departments VALUES (260, 'Shoe')");
stmt.addBatch("INSERT INTO emp_dept VALUES (1000, 260)");

// Submit a batch of update commands for execution
int[] updateCounts = stmt.executeBatch();
```

12.3.3 Improved security for DB2 authentication

Authentication security has been improved significantly with the DB2 Universal Driver for SQLJ and JDBC. It supports the following authentication techniques:

- User ID and password in plain text
- User ID and password encrypted
- Kerberos security

For encrypted and Kerberos security, the DB2 Universal Driver for SQLJ and JDBC uses the following Java services:

- IBM Java Generic Security Service (JGSS)
- IBM Java Authentication and Authorization Service (JAAS)
- IBM Java Cryptography Extension (JCE)

The authentication technique can be specified by either setting a Java property in the application, or by recording the required technique in the DB2 DataSource definition.

12.3.4 Improved Java SQL error information

The standard JDBC SQLException class does not provide a way to retrieve vendor-specific error information (other than the `getErrorCode()` method, which, for DB2, returns the SQLCODE reported by the DB2 server). For example, if an INSERT into a table fails because you try to insert a NULL value into a NOT NULL column, there is no standard way to find out which column DB2 is complaining about (other than to parse the error message).

Again, the DB2 Universal Driver for SQLJ and JDBC provides a proprietary API to retrieve detailed error information (in DB2 terms, to access the SQLCA).

This feature is discussed in more detail in "DB2 specific error handling" on page 242.

12.3.5 Java API for Set Client Information (SQLESETI)

As a DB2 specific extension, the JCC driver provides an interface to supply extended client information to the DB2 server. This is especially useful in a client/server or application server environment. Using this interface, the client program, or the application server, can supply information that identifies the specific user issuing a request. Otherwise, only the application server's information is passed. Unlike the user information supplied when creating the JDBC connection, the extended information can be changed at any time (for example, when the application server processes a request from another user).

Also, the extended client information can be used for accounting or workload management purposes.

The client information will be sent to the server at the next opportunity; that is, along with the next SQL call on that connection.

To use this interface, cast the `java.sql.Connection` object to a `com.ibm.db2.jcc.DB2Connection`. In addition to the standard `java.sql.Connection` methods, the `DB2Connection` class provides the following methods to set extended client information:

- setDB2ClientUser

  ```
  public abstract void setDB2ClientUser(String s)
    throws SQLException;
  ```

  Sets a user name for the connection. Unlike the user ID supplied when creating the connection, this can be a full user name.

- setDB2ClientWorkstation

  ```
  public abstract void setDB2ClientWorkstation(String s)
    throws SQLException;
  ```

  Sets a client workstation name for the connection, for example, the workstation's TCP/IP host name.

- setDB2ClientApplicationInformation

  ```
  public abstract void setDB2ClientApplicationInformation(String s)
    throws SQLException;
  ```

  Sets an application name. For example, you can specify the main class name of the Java application working with the connection.

- setDB2ClientAccountingInformation

  ```
  public abstract void setDB2ClientAccountingInformation(String s)
    throws SQLException;
  ```

  Sets accounting information.

Example 12-3 demonstrates the use of the extended client information API.

Example 12-3 Using the extended client information API

```
public class ClientInfoTest {

  public static void main(String[] args) {
    String url = "jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y";
    try {
      Class.forName("com.ibm.db2.jcc.DB2Driver");
      Connection conn = DriverManager.getConnection(url, "bartr1", "secret");
      if (conn instanceof DB2Connection) {
        DB2Connection db2conn = (DB2Connection) conn;
        db2conn.setDB2ClientUser("Ulrich Gehlert");
        db2conn.setDB2ClientWorkstation(java.net.InetAdress.getLocalhost().getHostName());
        db2conn.setDB2ClientApplicationInformation("ClientInfoTest");

        // Dummy call to force extended client information to be sent
        conn.prepareStatement("SELECT * FROM SYSIBM.SYSDUMMY1"
                              + "WHERE 0 = 1").executeQuery();
```

```
        // Sleep for a while 30 secs) so we can display DB2 thread information
        Thread.sleep(30000);
      }
    } catch (Throwable e) {
      e.printStackTrace();
    }
  }
}
```

Without setting client information, the DB2 DISPLAY THREAD command only shows the authorization ID and location name for the thread associated with the JDBC connection (see Figure 12-1).

```
 -DISPLAY THREAD(SERVER) DETAIL

DSNV401I  -DB7Y DISPLAY THREAD REPORT FOLLOWS -
 DSNV402I  -DB7Y ACTIVE THREADS -
 NAME     ST A   REQ ID           AUTHID   PLAN     ASID TOKEN
 SERVER   RA *     0 db2jccmain   BARTR1   DISTSERV 0050  1170
  V445-G9012728.G5AA.00F199927719=1170 ACCESSING DATA FOR 9.1.39.40
  V447--LOCATION        SESSID          A ST TIME
  V448--9.1.39.40       33756:1450      W R2 0232218533048
```

Figure 12-1 DISPLAY THREAD command showing basic thread information

After supplying client information, the output also shows the supplied user name, application name, and workstation name (see Figure 12-2).

```
-DISPLAY THREAD(*) DETAIL

DSNV401I  -DB7Y DISPLAY THREAD REPORT FOLLOWS -
 DSNV402I  -DB7Y ACTIVE THREADS -
 NAME     ST A   REQ ID           AUTHID   PLAN     ASID TOKEN
 SERVER   RA *     4 db2jccmain   BARTR1   DISTSERV 0050  1166
  V437-WORKSTATION=a23wpm64, USERID=Ulrich Gehlert,
       APPLICATION NAME=ClientInfoTest
  V445-G9012728.G5A4.00F1996EB1A2=1166 ACCESSING DATA FOR 9.1.39.40
  V447--LOCATION        SESSID          A ST TIME
  V448--9.1.39.40       33756:1444      W R2 0232218144291
```

Figure 12-2 DISPLAY THREAD command showing extended client information

Tip: Instead of setting the client information from your program, you can also set it up in the connection URL (or in a DataSource definition) using the extended URL syntax described in “Setting connection properties in the URL” on page 222 and the respective property keys listed in Table 12-1 on page 223.

12.3.6 Java API for application monitoring

To help you isolate performance problems with your applications, the DB2 Universal Driver for SQLJ and JDBC provides a proprietary API (DB2SystemMonitor class) to enable application monitoring.

The driver collects the following timing information, as depicted in Figure 12-3:

- Server time (the time spent in DB2 itself)
- Network I/O time (the time used to flow the DRDA protocol stream across the network)
- Core driver time (the time spent in the driver; this includes network I/O time and server time)
- Application time (the time between the start() and stop() calls)

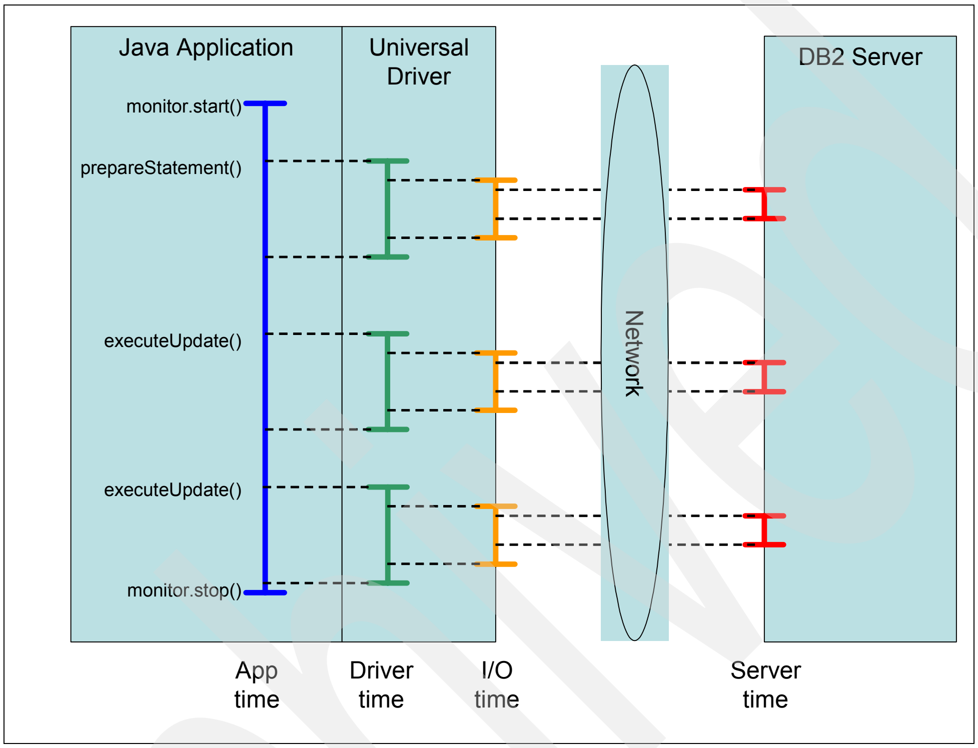

Figure 12-3 Times collected by DB2 Universal Driver for SQLJ and JDBC monitoring

The use of the API is demonstrated in Example 12-4.

Example 12-4 Using the DB2 Universal Driver for SQLJ and JDBC monitoring API

```
import com.ibm.db2.jcc.DB2Connection;
import com.ibm.db2.jcc.DB2SystemMonitor;

...
DB2SystemMonitor monitor = conn.getDB2SystemMonitor();
monitor.enable(true);
monitor.start(DB2SystemMonitor.RESET_TIMES);

// ... SQL statements ...

monitor.stop();
   // ServerTimeMicros is red line in figure above
System.out.println("Server time:      " + monitor.getServerTimeMicros());
   // NetworkIOTimeMicros is orange line in figure above
System.out.println("Network I/O time: " + monitor.getNetworkIOTimeMicros());
   // CoreDriverTimeMicros is green line in figure above
System.out.println("Core driver time: " + monitor.getCoreDriverTimeMicros());
   // ApplicationTimeMillis is blue line in figure above
System.out.println("Application time (ms): " + monitor.getApplicationTimeMillis());
```

You can choose to either reset or to accumulate times when starting the monitoring, using `DB2SystemMonitor.RESET_TIMES`, or `DB2SystemMonitor.ACCUMULATE_TIMES` on the `monitor.start()` method, respectively.

Note that the various getTime() methods may throw an `SQLException` if the driver cannot provide the information requested.

DB2SystemMonitor prerequisites

Although already shipped in an earlier FixPak, the DB2SystemMonitor is only fully operational with DB2 for LUW Version 8 FixPak 4.

In addition, to be able to get accurate times for the core driver and network I/O times, your JVM has to be at a certain code level as well. Currently these JVM enhancements have only been implemented as a feature of IBM's JVMs, more specifically in:

- JDK 131 SR 5, available now, for non-z/OS platforms
- JDK131 (cm131s), and JDK 141 SR1(cm141) for the z/OS platform

At the time of writing of this publication, the Sun JVM 1.4.1 does not have this feature. If the `DB2SystemMonitor` does not find the accurate timer functionality in the JVM, it throws an `SQLException`:

```
com.ibm.db2.jcc.a.SqlException: Network IO time in microseconds is not available
```

Or:

```
com.ibm.db2.jcc.a.SqlException: Core driver time in microseconds is not available
```

The ServerTime is based on the DRDA server elapsed time feature that was introduced in DB2for z/OS and OS/390 Version 7. As a consequence, DRDA has to be involved in the transaction to get the ServerTime. If you are running your application local to DB2, you always get 0 (zero) from the getServerTimeMicros method. This is true both for DB2 LUW and DB2 for z/OS when running the JCC Driver as a Type 2 driver. When run as a Type 4 driver, JCC always uses DRDA to connect to the server.

Please also note that the ApplicationTime is in milliseconds (using System.currentTimeMillis), wheras the other times are presented in microseconds.

12.3.7 Native DB2 server SQL error messages

Each DB2 server, including DB2 for OS/390 V6 and V7, now provides stored procedures that allow applications to retrieve the "native" error message text for a given error or warning. The name of the stored procedure is SYSIBM.SQLCAMESSAGE. See "Required DB2 for z/OS changes to enable the Universal Driver" on page 58.

12.3.8 Multiple open cursors

Currently (DB2 for z/OS and OS/290 Version 7) it is not possible for an application to have more than one instance of an open cursor. An attempt to open a cursor that is already open fails with SQLCODE -502. DB2 returns an error because the cursor is already open from the previous call.

Although this may be OK for most cursors in embedded applications, it poses a big problem for SQLJ iterators. The SQLJ API syntax allows an application to issue a "new" operation, to make a new copy of the static cursor that can be used with different host variable input at

OPEN time. Since a cursor currently has to be unique based on the fully qualified package name, consistency token, and section number, it is not possible for an SQLJ application to have more than one instance of an open cursor. A second instance or OPEN of the same cursor would not be allowed in DB2, and results in an SQLCODE -502.

DB2 for z/OS Version 8 allows multiple opens for the same cursor used by an SQLJ iterator. The DB2 server now provides a unique identifier to the requester for each open cursor or result set. The request can then manage the multiple instances using the unique cursor identifier.

Embedded SQL applications (non-SQLJ applications) will not be allowed to take advantage of this new processing.

12.3.9 SAVEPOINT support

The DB2 Universal Driver for SQLJ and JDBC supports the SAVEPOINT mechanism specified in the JDBC 3.0 specification. For further details about savepoints, refer to the JDBC 3.0 specification.

12.3.10 Auto-generated keys

Like many other database servers, DB2 has a mechanism that allows you to automatically generate a new, unique key value whenever a row is inserted. In the case of DB2, you declare a column to be an IDENTITY column.

Of course, after inserting a new row into a table containing an IDENTITY column, you probably want to retrieve the value that DB2 generated for that column (you may need it, for example, as a foreign key value in a dependant table). You can use the DB2-specific function `IDENTITY_VAL_LOCAL()` to retrieve the last value generated.

Beginning with JDBC 3.0, however, there is a mechanism that allows an application to retrieve the value without using vendor-specific extensions, and this API is supported by the DB2 Universal Driver for SQLJ and JDBC.

Example 12-5 shows how to use this API. For further details refer to Section 13.6 of the JDBC 3.0 API specification. The JDBC 3.0 specification can be found on the Web at:

http://java.sun.com/products/jdbc/download.html

Example 12-5 Retrieving auto-generated keys

```
Statement stmt = conn.createStatement();
// indicate that the key generated is going to be returned
int rows = stmt.executeUpdate("INSERT INTO ORDERS (ISBN, CUSTOMERID) “
                            + "VALUES (195123018, ’BILLG’)",
                              Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
if (rs.next()) {
// retrieve the new key value
  int orderID = rs.getInt(1);
}
```

Note: Unfortunately, there is no equivalent construct for SQLJ; you have to use the IDENTITY_VAL_LOCAL() function:

```
#sql { SELECT IDENTITY_VAL_LOCAL() INTO :orderID FROM SYSIBM.SYSDUMMY1 };
```

In DB2 for z/OS Version 8, you will be able to use the 'SELECT FROM INSERT' construct to retrieve generated values of a table:

```
#sql { SELECT order-identity-col INTO :orderID
                   FROM FINAL TABLE (INSERT INTO my-order-table(col-list)
                                     VALUES(column-values) )
               };
```

13

Performance topics

In this chapter we discuss several hints and tips for optimizing the performance of JDBC and SQLJ applications.

We discuss the following:

- General performance recommendations, applying to both JDBC and SQLJ
- SQLJ performance considerations
- System-level performance tuning

13.1 General performance recommendations

Let us start by looking at over some general Java-DB2 recommendations that will help you to obtain better performance.

13.1.1 Use static SQL wherever possible

Although dynamic statement caching (see "Turn on DB2 dynamic statement caching" on page 238) relaxes the situation somewhat, static SQL is generally much faster than dynamic SQL because the SQL parsing and access path calculation is done at compile time, rather than at runtime.

In some situations, however, JDBC can be the better choice, not from a performance viewpoint, but from a functionality point of view. For example, a GUI application may allow the user to construct queries interactively, with a large number of options that just do not allow every possible query to be coded statically.

13.1.2 Turn auto commit off

By default, when you open a database connection via the `DriverManager` class, that connection has the autoCommit property set to true, which forces a commit after every single SQL statement. You should turn auto commit off, and commit only when appropriate.

13.1.3 Only retrieve/update columns as needed

Column processing is one of the major factors in CPU resource consumption. The two primary reasons for this are:

- Strings for character string columns must be converted between Unicode (Java) and EBCDIC/ASCII (DB2 engine).
- A Java object is created per column per row for those data types that are not primitive types in Java, such as character string columns.

So, in order to reduce this overhead, you should only retrieve or update the columns you need.

13.1.4 Store numbers as numeric data types

This may be obvious but is still sometimes done wrong. Use the DB2 numeric data types, not CHARACTER, for numeric data. For example, consider storing ZIP codes in an INTEGER column rather than in a CHARACTER(5) column.

This saves the overhead of creating an object (if the column is declared NOT NULL), and of EBCDIC/Unicode conversion.

13.1.5 Use DB2 built-in functions

DB2 comes with many useful built-in functions that are more efficient than their Java counterparts. For example, when retrieving a fixed-width character data column, you may want to get rid of the trailing blanks that DB2 appends whenever the value is shorter than the column's length. You can use the Java `String.trim()` method, like this:

```
#sql { SELECT JOB INTO :job FROM DSN8710.EMP ... };
job = job.trim();
```

However, it is more efficient (and easier) to use the DB2 TRIM function since no intermediate String object has to be created.

```
#sql { SELECT TRIM(JOB) INTO :job FROM DSN8710.EMP ... };
```

13.1.6 Release resources

Another thing that developers sometimes forget to do is to close and release resources when they are no longer being used. The JDBC driver maintains its own links to resources, and the resources are only released when the resources are closed, or when the connection is closed. For this reason:

- Close ResultSets when the application is done with them. If you do not do this, JVM garbage collection cannot reclaim the objects, and eventually the application may run out of JDBC resources or, even worse, run out of memory.
- Close PreparedStatements that have ResultSets as soon as they are done being used. Closing the ResultSet is not enough to release the underlying cursor resource. If you do not close the PreparedStatement, the cursor resource is tied up for the life of the PreparedStatement.
- Close CallableStatements when you are done with them, too, or else the application may run out of call sections.
- Be sure to release resources even in the case of failure. The Java try / finally construct is well suited to achieve this (see Example 14-3 on page 242).

SQLJ makes things a bit easier for developers than JDBC, because the SQLJ translator automatically generates the code to release statements. However, you still have to close Iterators yourself.

13.2 JDBC recommendation

JDBC applications should call prepared statement setters that match the column type at the server. This helps especially in cases where deferred prepare is on, and the driver has to guess the data type to send to the server. If the driver guesses incorrectly, an error may be returned from the server. In some cases the driver will attempt a retry of the statement using a different data type. This results in two network flows and defeats the advantage of having deferred prepare enabled in the driver.

13.3 SQLJ performance considerations

The following sections discuss performance topics that are specific to SQLJ applications, namely:

- Use of matching data types
- Positioned iterators vs. named iterators
- Importance of online checking
- Importance of explain table
- Rebinding your applications

13.3.1 Use matching data types

Use the recommended mappings of DB2 to Java data types. For example, while it is perfectly legal to FETCH a TIMESTAMP column into a String variable, you definitely should not do so.

For one thing, using a String is less efficient since the SQLJ runtime has to format the TIMESTAMP column into String format. Additionally, using a java.sql.Timestamp variable allows you to control the formatting of the timestamp yourself.

Also, as explained in “Always customize with online checking enabled” on page 236 below, a non-matching data type may result in a poor access path if that host variable is part of a predicate.

13.3.2 Use positioned iterators, not named iterators

Named iterators are actually implemented as a wrapper around positioned iterators, with an associated hash table that maps column names to column numbers. For best performance, we recommend that you use positioned iterators that do not have this overhead.

13.3.3 Always customize with online checking enabled

Online checking is especially important when predicates using host variables are involved. As explained in “Use matching data types” on page 235 above, host variables should match their corresponding columns in data type and size.

In order for a predicate to use a matching index scan, the definition in the Java package must match the definition in the DB2 catalog, both in terms of data type and length. Because Java String objects do not have a concept of length, that information can only be obtained from the catalog. This is part of online checking process.

Suppose you have the following SQL statement:

```
#sql {
  SELECT NAME
    FROM SYSIBM.SYSTABLES
   WHERE CREATOR = :creator
};
```

Example 13-1 shows an excerpt of the Explain table after two binds of the generated serialized profile, one with online checking enabled, the other one with online checking disabled. While the first one uses an index, the second does a table space scan.

Example 13-1 Result of bind with online checking enabled vs. disabled

```
SELECT PROGNAME, ACCESSTYPE, MATCHCOLS, ACCESSCREATOR, ACCESSNAME
  FROM PLAN_TABLE
WHERE PROGNAME = 'IXTEST2';
---------+---------+---------+---------+---------+---------+------
PROGNAME  ACCESSTYPE  MATCHCOLS  ACCESSCREATOR  ACCESSNAME
---------+---------+---------+---------+---------+---------+------
IXTEST2   I                  1  SYSIBM        DSNDTX01                 1
IXTEST2   R                  0                                         2
DSNE610I NUMBER OF ROWS DISPLAYED IS 2
```

Notes on Example 13-1:

1. Result of bind with online checking enabled. The package uses index SYSIBM.DSNDTX01.
2. Result of bind with online checking disabled. The package does a tablespace scan (ACCESSTYPE = R).

Note that not only CHARACTER columns can be affected, but also numeric columns. If you use a host variable of type long to match a column of type INTEGER, the optimizer will choose a non-matching index scan because the predicate has to be evaluated at Stage 2 rather than at Stage 1.

13.3.4 Check explain tables

As discussed in the previous section, it is a good idea to always bind with EXPLAIN(YES) and to check the PLAN_TABLE for potential performance problems. See Chapter 26 of the *Application Programming and SQL Guide*, SC26-9933, for information about how to set up and interpret a PLAN_TABLE.

Alternatively or additionally, you can use Visual Explain, which is a non-priced feature of DB2 for OS/390 (in other words, it is free). It lets you graphically analyze the access paths that DB2 chooses, which eliminates the need to manually interpret the plan_table output. You can download Visual Explain at:

`http://www.ibm.com/software/data/db2/os390/db2ve/`

13.3.5 Rebind packages regularly

You should rebind your packages after administrative tasks that may affect program performance, such as a REORG, or the creation of a new index, or after a significant amount of INSERT/UPDATE activity.

13.4 System-level performance tuning

This section briefly summarizes several system level performance recommendations collected by IBM Silicon Valley Lab experts.

For a more in-depth discussion of the various DB2 parameters that influence dynamic SQL performance, see *Squeezing the Most Out of Dynamic SQL with DB2 for z/OS and OS/390,* SG24-6418.

13.4.1 Tune the JVM heap size

In a Java database workload using either JDBC or SQLJ to access relational data, a lot of Java objects are created and then destroyed. The JVM heap size plays an important role in overall Java application performance. The default initial heap size is 1 MB, and the default maximum heap size is 8 MB. In almost all cases these default sizes are not sufficient and lead to poor performance. Studies at the IBM Silicon Valley Lab and at customer installations have shown that setting the heap size and the maximum heap size to an equal large value increases the throughput dramatically. This is because scanning for garbage collection is not triggered so often, thereby reducing the repeated scanning of long living objects. 300–400MB heap sizes are not uncommon.

13.4.2 Get the latest code and maintenance

Keep current with JDK releases and with upgrades to the JDBC driver. While the initial development emphasis was on delivering JDBC function, in the meantime a lot of work has been done regarding improvements in CPU performance by reducing column processing overheads. This has been delivered through improvements to the JDBC driver and the JDK/JVM. The JDBC 2.0 driver delivered with DB2 Version 7 has been the base for delivering these performance enhancements.

13.4.3 Turn on DB2 dynamic statement caching

Turn on DB2 dynamic SQL statement caching with CACHEDYN=YES in your subsystem parameters (DSNZPARM). This option causes dynamically prepared SQL statements to be cached across transaction boundaries, and can therefore dramatically improve the performance of JDBC applications, especially when the same, limited set of SQL statements is executed over and over again, and parameter markers are used.

14

Error handling and diagnosis

This chapter discusses error handling, extended debugging, and tracing facilities. In particular, we provide a description of the DB2-specific extensions to aid you with diagnosis, tracing, and accounting.

14.1 Basic error handling

As with all error handling in Java, error handling for JDBC and SQLJ uses the `try / catch` construct. Whenever an error is encountered in a JDBC or SQLJ program, the JDBC driver throws an `SQLException`. JDBC provides getter methods to produce the SQLCODE and the related message. These methods are `getErrorCode()` to return the SQLCODE and `getMessage()` to produce the full text error message related to the error generated. Both are methods of the `SQLException` class.

> **Tip:** Although the objects and methods described here are JDBC related, they also apply to SQLJ programs. In fact, error handling for JDBC and SQLJ programs is identical.

Warning messages are handled differently and do not throw an exception so, if desired, they must be handled specifically using the `SQLWarning` class. As multiple warnings can be generated from a single SQL statement, JDBC creates an object for each warning received and chains them together.

If a program is to output the warnings it receives, the `SQLWarning` objects have to be retrieved using the method `getWarnings()` of the `Connection` object. You then step through the warnings using the `getNextWarning()` of the `SQLWarning` object. Like the `SQLException` class, the `SQLWarning` class allows you to retrieve the SQLSTATE and SQLCODE, using the methods `getSQLState()` and `getErrorCode()`, respectively.

Example 14-1 gives examples of error and warning handling in JDBC.

Example 14-1 Error handling in JDBC

```
try {
  System.out.println("*** Connecting to DB2 for OS/390 database ***");

  con = DriverManager.getConnection("jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y",
                                    "sflynn",
                                     "girlie01");

  System.out.println("*** Connection made ***");

  stmt = con.createStatement();
  System.out.println("*** Statement created ***");

  rs = stmt.executeQuery(
         "SELECT STRIP(CREATOR) AS CREATOR, NAME, RECLENGTH " +
         "FROM SYSIBM.SYSTABLES " +
         "WHERE DBNAME = 'DSNDB06' " +
         "  AND TYPE = 'T' " +
         "ORDER BY 2, 1");
  System.out.println("*** Statement executed ***");

  // Print all warnings
  for (SQLWarning warn = stmt.getWarnings(); warn != null; warn = warn.getNextWarning()) {
    System.out.println("*** Statement warning code: " + warn.getErrorCode());
  }

  rs.close();
  stmt.close();
  con.close();
} catch(SQLException e) {
  System.out.println("SQLCODE = " + e.getErrorCode());
  System.out.println("SQLMESSAGE = " + e.getMessage());
```

```
  e.printStackTrace();
} catch(Exception e) {
  e.printStackTrace();
}
```

14.2 SQLCODE and SQLSTATE

The `SQLException` class has two methods to retrieve an error code associated with an exception:

- The `getErrorCode()` method, which returns a vendor-specific error code as an integer. In DB2, this vendor-specific error code is the SQLCODE.
- The `getSQLState()` method, which returns a five-character string, the SQLSTATE.

If you are interested in maximum portability, we recommend that you use the `getSQLState()` method rather than the `getErrorCode()` method for error handling.

The SQLSTATE has another advantage. Whereas the SQLCODE values are somewhat randomly scattered, the SQL state is designed so that applications can easily check for a category of errors. The first two characters of the five-character string denote the error category, and the remaining three characters denote a specific error within that category. Appendix A, “SQLSTATE categories” on page 291, summarizes the SQL state categories.

14.3 Cleaning up resources

Even in the case of an exception, a well-behaved program should take care to free all acquired resources.

In Example 14-2, an iterator is opened and used to iterate over the result set from a query. When the entire result set has been processed, the iterator is closed. Now suppose an `SQLException` (or any other exception, for that matter) is thrown in the loop. The call to iter.close() will never be performed, and the iterator will remain open until the current transaction ends. This has two negative impacts:

- Depending on the execution environment, there is a limited number of iterators that can be open at the same time. Suppose the `foo()` method in Example 14-2 is called from a loop, and foo() always throws an exception in its loop body, you will find the program running out of available iterators very soon.
- Unless your application runs with uncommitted read isolation, the row or page the iterator is currently positioned on is likely to be locked (unless lock avoidance is used). Other applications will not be able to update the row, or even to read it if the iterator has positioned UPDATE enabled.

Example 14-2 Bad style - Iterator not closed in case of exception

```
// NOT RECOMMENDED -- iterator not closed in case of exception
void foo() throws SQLException {
  MyIterator iter;
  #sql iter = { SELECT ... FROM ... };
  while (true) {
    #sql { FETCH :iter INTO ... };
    if (iter.endFetch()) break;
    ...
  }
  iter.close();
```

```
}
```

In Example 14-3, we use the `try / finally` construct of Java to ensure that the iterator will always be closed, even when an exception occurred. The Java language specification guarantees that a finally block is always executed, no matter whether the corresponding try block completes successfully or raises an exception.

Example 14-3 try / finally construct to ensure proper closing of iterator

```
void foo() throws SQLException {
  MyIterator iter = null;
  try {
    #sql iter = { SELECT ... FROM ... };
    while (true) {
      #sql { FETCH :iter INTO ... };
      if (iter.endFetch()) break;
    }
  } finally {
    if (iter != null) iter.close();
  }
}
```

Be especially careful when using holdable iterators (see “Holdable iterators” on page 186). Holdable iterators remain open until you explicitly close them, or a ROLLBACK occurs.

14.4 DB2 specific error handling

Suppose you received an `SQLException` that says that you tried to insert a null value into a NOT NULL column, and you want to figure out which column it was that DB2 complained about. The standard `SQLException` class does not provide an interface to retrieve vendor-specific information (other than `getErrorCode()`, which, for DB2, returns the SQLCODE). However, the DB2 JCC driver provides a proprietary method to examine the SQLCA, which is a DB2 data structure providing information about the execution of SQL statements.

To access the SQLCA, you cast the `SQLException` object to `com.ibm.db2.jcc.DB2Diagnosable` and call the `getSqlca()` method to retrieve an instance of class `com.ibm.db2.jcc.DB2Sqlca`. This class, in turn, provides methods to access the individual fields of the SQLCA. The meaning of each field depends on the specific kind of error.

The most interesting part of the SQLCA is a string called SQLERRM, which contains several error tokens, separated by the character 0xFF. You do not need to tokenize this string yourself; the `DB2Sqlca.getSqlErrmcTokens()` method does this for you.

To find out what the individual error tokens mean for a given SQLCODE, refer to *DB2 Universal Database for OS/390 and z/OS Messages and Codes*, GC26-9940. Look up the error text in the description of the SQLCODE. The tokens appear sequentially in the SQLERRM in the order they appear in the message text.

For example, this is the error text for SQLCODE -805:

-805 **DBRM OR PACKAGE NAME** *location-name.collection-id.dbrm-name.consistency-token* **NOT FOUND IN PLAN** *plan-name*. **REASON** *reason*

So, when you receive a -805 SQLCODE, `DB2Sqlca.getSqlErrmcTokens()` will return a six-element array containing the location name, collection ID, DBRM name, consistency token, plan name, and reason code, in that order. The meaning of the reason code is also documented in the message description.

Example 14-4 demonstrates how to retrieve DB2-specific error information.

Example 14-4 Retrieving DB2-specific extended error information

```
try {
  ... SQL calls ...
} catch (SQLException e) {
  if (e instanceof DB2Diagnosable) {
    DB2Sqlca sqlca = ((DB2Diagnosable) e).getSqlca();
    System.err.println("SQLERRM:  " + sqlca.getMessage());
    System.err.println("SQLCODE:  " + sqlca.getSqlCode());
    System.err.println("SQLSTATE: " + sqlca.getSqlState());
    String[] tokens = sqlca.getSqlErrmcTokens();
    if (tokens != null) {
      for (int i = 0; i < tokens.length; i++)
        System.err.println("ERRMC[" + i + "]: " + tokens[i]);
        System.err.println("SQLWARN: " + new String(sqlca.getSqlWarn()));
      }
    }
  else {
    System.err.println(e);
    }
  }
```

In addition, the DB2 JCC driver provides the utility class `com.ibm.db2.jcc.DB2ExceptionFormatter` which prints formatted error information to a stream (similar to what we did explicitly in 14-4). You may find this utility class useful for logging purposes.

Summary of DB2Sqlca methods

Below is a summary of DB2Sqlca methods:

- getSqlErrmcTokens

  `public abstract String[] getSqlErrmcTokens();`

  Returns the error message tokens as described by the DB2 SQLCA sqlerrmc field. Each element in the returned String[] contains one of the sqlerrmc tokens. For example, if you did not supply a value for a NOT NULL column in an INSERT statement, the first (and only) element in the array will contain the column name. A null value is returned if the value of the sqlerrmc field is not relevant.

- getSqlErrmc

  `public abstract String getSqlErrmc();`

  Returns the error message tokens as a String delimited with semicolons. A null value is returned if the value of the sqlerrmc field is not relevant.

- getSqlErrp

  `public abstract String getSqlErrp();`

  Returns the name of the DB2 module that issued the error, or null.

- getSqlErrd

  `public abstract int[] getSqlErrd();`

Returns error information as described by the DB2 SQLCA sqlerrd field.

- getSqlWarn

  `public abstract char[] getSqlWarn();`

  Returns warning information as described by the DB2 SQLCA sqlwarn field, as a fixed size char[11] array.

- getSqlCode

  `public abstract int getSqlCode();`

  Returns the SQLCODE (same as `SQLException.getErrorCode()`).

- getSqlState

  `public abstract String getSqlState ();`

  Returns the SQL state (same as `SQLException.getSQLState()`).

- getMessage

  `public abstract String getMessage();`

  Returns the server error message text (same as `SQLException.getMessage()`).

Remember also that you can get a nicely formatted error message using the `SQLException.getMessage()` method, similar to what you have today in DB2 for z/OS when you call DSNTIAR to format you SQL error information. It does require that you have the `retrieveMessagesFromServerOnGetMessage` property set on your connection. An example is shown in Example 6-1 on page 103.

14.5 Tracing

A very useful feature of the DB2 Universal Driver for SQLJ and JDBC is its tracing support, which can be turned on either from your program, or even externally by setting up properties on a DataSource definition, or in the JDBC connection URL. This way, you can turn on tracing even when running an application for which you do not have the source code, provided that the connection URL is specified externaly to the program.

14.5.1 Turning on tracing in the program

To turn on tracing programatically, use the `setJccLogWriter()` method of class `DB2Connection`. The first argument to this method is a `PrintWriter` to which the output is sent. The optional second argument specifies the trace level. Table 14-1 lists the available trace levels. The constants representing these levels are declared in class `com.ibm.db2.jcc.DB2BaseDataSource`. You can combine individual levels using bitwise OR. If you use the one-argument `setJccLogWriter()` method, `TRACE_ALL` is assumed.

Table 14-1 Trace level options for the DB2 Universal Driver for SQLJ and JDBC

Constant name	Value
TRACE_NONE	0x0000
TRACE_CONNECTION_CALLS	0x0001
TRACE_STATEMENT_CALLS	0x0002
TRACE_RESULT_SET_CALLS	0x0004
TRACE_DRIVER_CONFIGURATION	0x0010

Constant name	Value
TRACE_CONNECTS	0x0020
TRACE_DRDA_FLOWS	0x0040
TRACE_RESULT_SET_META_DATA	0x0080
TRACE_PARAMETER_META_DATA	0x0100
TRACE_DIAGNOSTICS	0x0200
TRACE_SQLJ	0x0400
TRACE_XA_CALLS	0x0800
TRACE_ALL	0xFFFFFFFF

To turn tracing off, call `setJccLogWriter()` with a `null` argument. Example 14-5 shows how to turn a trace on and off.

Example 14-5 Turning on tracing from your program

```
DB2Connection conn = (DB2Connection) DefaultContext.getDefaultContext().getConnection();
PrintWriter jccLogWriter = new PrintWriter(new FileWriter("C:/temp/jcctrace.log"));
int traceLevel = DB2BaseDataSource.TRACE_CONNECTION_CALLS
               | DB2BaseDataSource.TRACE_DRIVER_CONFIGURATION
               | DB2BaseDataSource.TRACE_RESULT_SET_CALLS
               | DB2BaseDataSource.TRACE_STATEMENT_CALLS
               ;
conn.setJccLogWriter(jccLogWriter, traceLevel);   // Turns tracing on
...
conn.setJccLogWriter(null);                        // Turns tracing off
```

The trace produced by running the Hello sample from Getting started with SQLJ is listed in Example 14-6.

Example 14-6 Trace output from running the Hello application

```
[Thread:main][Connection@48f70faa] prepareStatement (SELECT LASTNAME, FIRSTNME, SALARY FROM
DSN8710.EMP WHERE SALARY BETWEEN ? AND ? ORDER BY 1, 2       , 1003, 1007) called
[Thread:main][Connection@48f70faa] prepareStatement () returned PreparedStatement@22df8fa9
[Thread:main][PreparedStatement@22df8fa9] setBigDecimal (1, 30000) called
[Thread:main][PreparedStatement@22df8fa9] setBigDecimal (2, 50000) called
[Thread:main][PreparedStatement@22df8fa9] executeQuery () called
[Thread:main][PreparedStatement@22df8fa9] executeQuery () returned ResultSet@4b7d0fa9
[Thread:main][ResultSet@4b7d0fa9] getMetaData () called
[Thread:main][ResultSet@4b7d0fa9] getMetaData () returned com.ibm.db2.jcc.c.j@4e1ecfa9
[Thread:main][PreparedStatement@22df8fa9] getWarnings () returned null
[Thread:main][ResultSet@4b7d0fa9] next () called
[Thread:main][ResultSet@4b7d0fa9] next () returned true
[Thread:main][ResultSet@4b7d0fa9] getString (1) called
[Thread:main][ResultSet@4b7d0fa9] getString () returned GEYER
[Thread:main][ResultSet@4b7d0fa9] getString (2) called
[Thread:main][ResultSet@4b7d0fa9] getString () returned JOHN
[Thread:main][ResultSet@4b7d0fa9] getBigDecimal (3) called
[Thread:main][ResultSet@4b7d0fa9] getBigDecimal () returned 40175.00
[Thread:main][ResultSet@4b7d0fa9] next () called
[Thread:main][ResultSet@4b7d0fa9] next () returned true
... rest of result set omitted ...
[Thread:main][ResultSet@4b7d0fa9] next () called
[Thread:main][ResultSet@4b7d0fa9] next () returned false
```

```
[Thread:main][Connection@48f70faa] commit () called
[Thread:main][Connection@48f70faa] close () called
```

Usually, TRACE_DRDA_FLOWS should be turned off since it generates a lot of output. For example, to enable all trace levels except TRACE_DRDA_FLOWS, you could simply code `~TRACE_DRDA_FLOWS` (~ is the bitwise NOT operator in Java).

14.5.2 Turning on tracing using connection properties

As we said before, tracing can also be turned on from outside your program, provided that the JDBC URL is not hard-coded into the program (it could be specified on the command line, in a properties file, or in a DataSource definition). Use the syntax and property keys described in "Setting connection properties in the URL" on page 222. To find out the value for the property key `traceLevel`, combine the trace levels you want enabled and convert the result to an integer. For example, to trace statement calls, result set meta data, and parameter meta data, use:

```
   TRACE_STATEMENT_CALLS | TRACE_RESULT_SET_META_DATA | TRACE_PARAMETER_META_DATA
= 0x0002 | 0x0080 | 0x0100 = 0x0182
= 386
```

So the URL looks like this:

```
jdbc:db2://your.server.name:port/SSID:traceFile=jcctrace.log:traceLevel=386
```

Again, the default for `traceLevel` is `TRACE_ALL` if you specified a trace file name but no trace level.

> **Tip:** When the trace file name contains colons, you have to enclose it in double quotes:
>
> `jdbc:db2://your.server.name:port/SSID:traceFile="C:/temp/jcctrace.log":traceLevel=386`

Part 4

Accessing DB2 from Web applications

In the previous chapters we showed you how to develop and run stand-alone Java applications for DB2. In this part we show you how to access DB2 from a Web server environment, using the Java Servlet and JavaServer pages technologies.

15

Using Servlets to access DB2

In this chapter we demonstrate how to access DB2 from a Web server environment. Since we do not want to reinvent the wheel, we use the code developed in Chapter 11, “SQLJ revisited” on page 199, using it from the Servlets that generate the dynamic HTML content.

The simple Web application we develop in this chapter displays a list of employees. Each employee serial number is a hyperlink that takes the user to a detail page, displaying the employee’s personal data and picture.

To develop and test the code, we use the WebSphere test environment of WSAD.

15.1 Creating the project

In WSAD, a Web application must live in a Web project. To create the new project, select **File -> New -> Project**. Select **Web**, then **Dynamic Web Project**, and click **Next**.

In the Project name field, type `SG24-6435-Web`. Make sure that the Configure advanced options box is selected and click **Next**.

On the next panel, enter `SG24-6435` in the Context root field, overriding the default value that is the project name. Make sure that J2EE level 1.3 is selected, then press **Finish**.

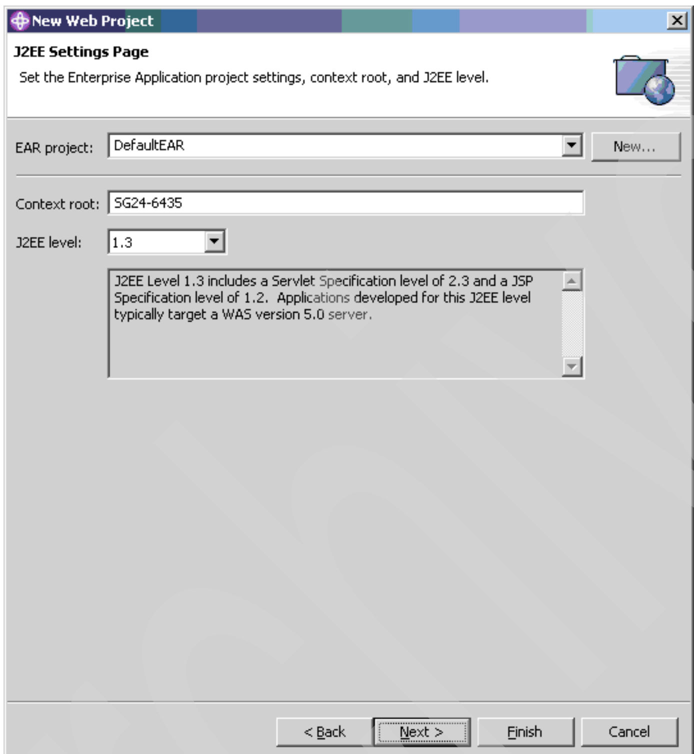

Figure 15-1 Creating a Web project (step 1)

WSAD now asks if you want to change to the Web perspective. Click **OK**. In the J2EE Navigator view, you will see three projects listed: Our original SG24-6435 project containing the Employee program; the SG24-6435-Web project; and a project called DefaultEAR, which was automatically created by WSAD. After expanding the SG24-6435 folder and its subfolders, your screen should look similar to Figure 15-2 on page 251.

Note: In WSAD, a Web project has to be associated with an EAR project, which in turn groups several individual Web projects (also called modules in this context) into a larger application. For the sake of this example, we do not need the EAR functionality, which is why we let WSAD create a dummy EAR project for us.

Notice that Web projects are organized in a slightly different way as opposed to Java projects. In Java projects, the compiled .class files by default go into the same directories as their corresponding source files, whereas in Web projects, the source and binary files are kept separate (the JavaSource and WebContent/WEB-INF/classes folders, respectively). This reflects the packaging structure of Web applications. Basically, everything under the WebContent folder is packaged together for deployment.

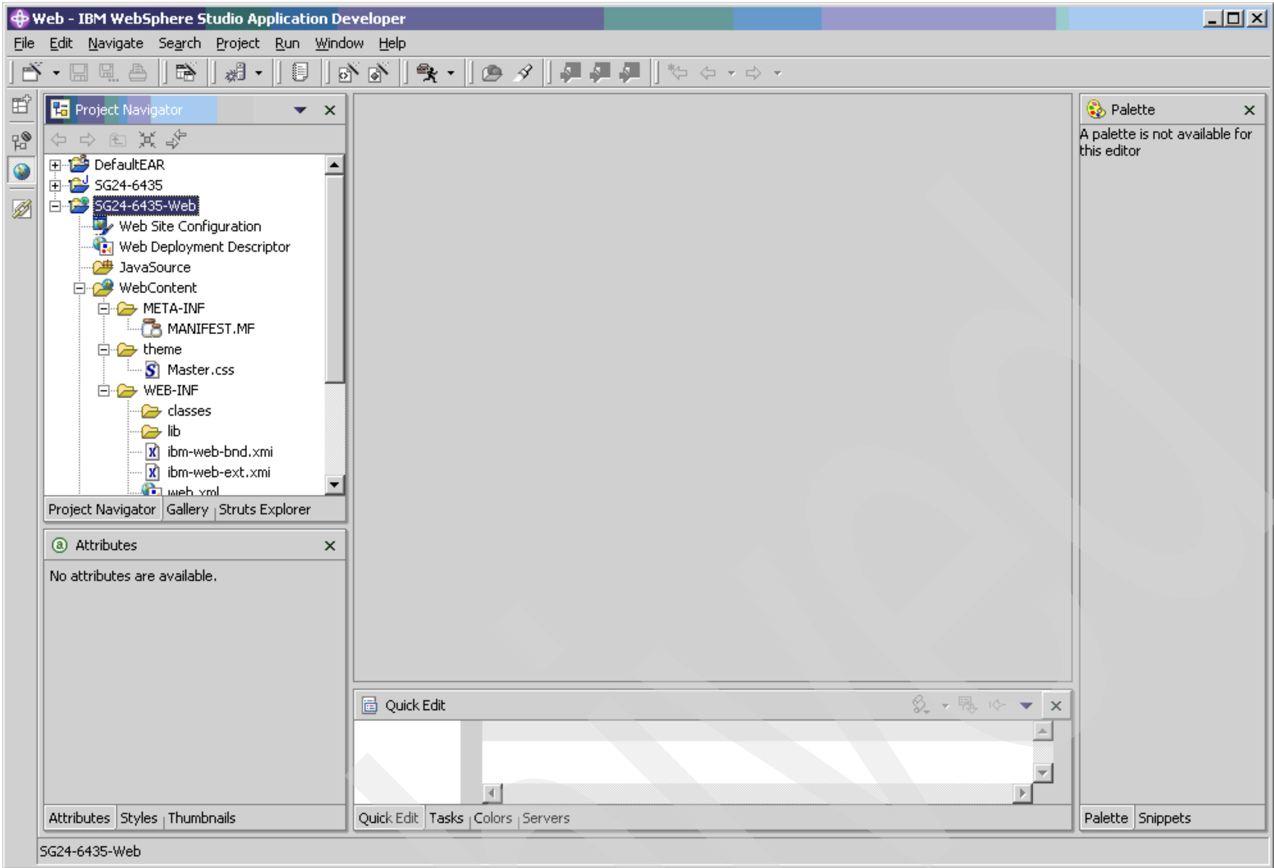

Figure 15-2 The WSAD Web perspective

15.2 Creating the EmployeeList Servlet

Now we are ready to create our first Servlet, called EmployeeListServlet. As the name suggests, it will (eventually) display the contents of the employee table.

In the code for the Servlet, we make use of the Employee class created in Chapter 11, “SQLJ revisited” on page 199.

We create the Servlet using the New Servlet wizard. Highlight the Java Source folder and select **File -> New -> Servlet**. Enter `com.ibm.itso.sg246435.web` for the Java package name, and `EmployeeListServlet` in the Class name field. Since our Servlet will generate an HTML page, the default superclass javax.servlet.http.HttpServlet is fine. Click **Next**, and unselect the doPost() check box, as we do not need a doPost() method in our application. Click **Finish** to create the Servlet. In the editor window that opens, you will see the stub source code, which contains only one method, called `doGet()`. This method is invoked by the Web server when a client makes a request to that Servlet.

15.2.1 Implementing the doGet() method

The implementation of the `doGet()` method as created by the New Servlet wizard does nothing except call the `doGet()` method provided by its superclass HttpServlet (which in turn sends an error code back to the client). So, we have to replace that default implementation.

We start with a skeleton Servlet that, for the time being, just displays a heading (see Example 15-1). You will have to add an import statement for class `java.io.PrintWriter` (not shown in the example code).

> **Tip:** A convenient way to have WSAD add the import statement for you is to use the Code Assist feature. Position your text cursor after the class name (`PrintWriter`) and press Ctrl+Space. WSAD automatically adds an import statement.

Example 15-1 The doGet() method skeleton

```
public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, IOException
{
  PrintWriter out = resp.getWriter();
  out.println("<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>");
  out.println("<HTML>");
  out.println("<HEAD>");
  out.println("<TITLE>Employee listing</TITLE>");
  out.println("</HEAD>");
  out.println("<BODY>");
  out.println("<H1>Employee listing</H1>");
  out.println("</BODY>");
  out.println("</HTML>");
  out.close();
}
```

15.2.2 Testing the Servlet

We are now ready to test the Servlet skeleton. Fortunately, we do not have to install and configure a full-blown Web application server in order to run the Servlet. WSAD comes with a component called the *WebSphere test environment*, which is basically a stripped-down version of WebSphere Application Server that has been integrated into the WSAD environment.

WSAD takes you through all the steps to create a WebSphere test environment. In the J2EE Navigator view, right-click the Servlet class and select **Run on Server…** (Figure 15-3 on page 253). Since no test environment has been created yet, WSAD will prompt you to create one. On the following dialog, select the **Create new server** radio button (if not selected by default), and select **WebSphere version 5.0 -> Test Environment**, then click **Next**.

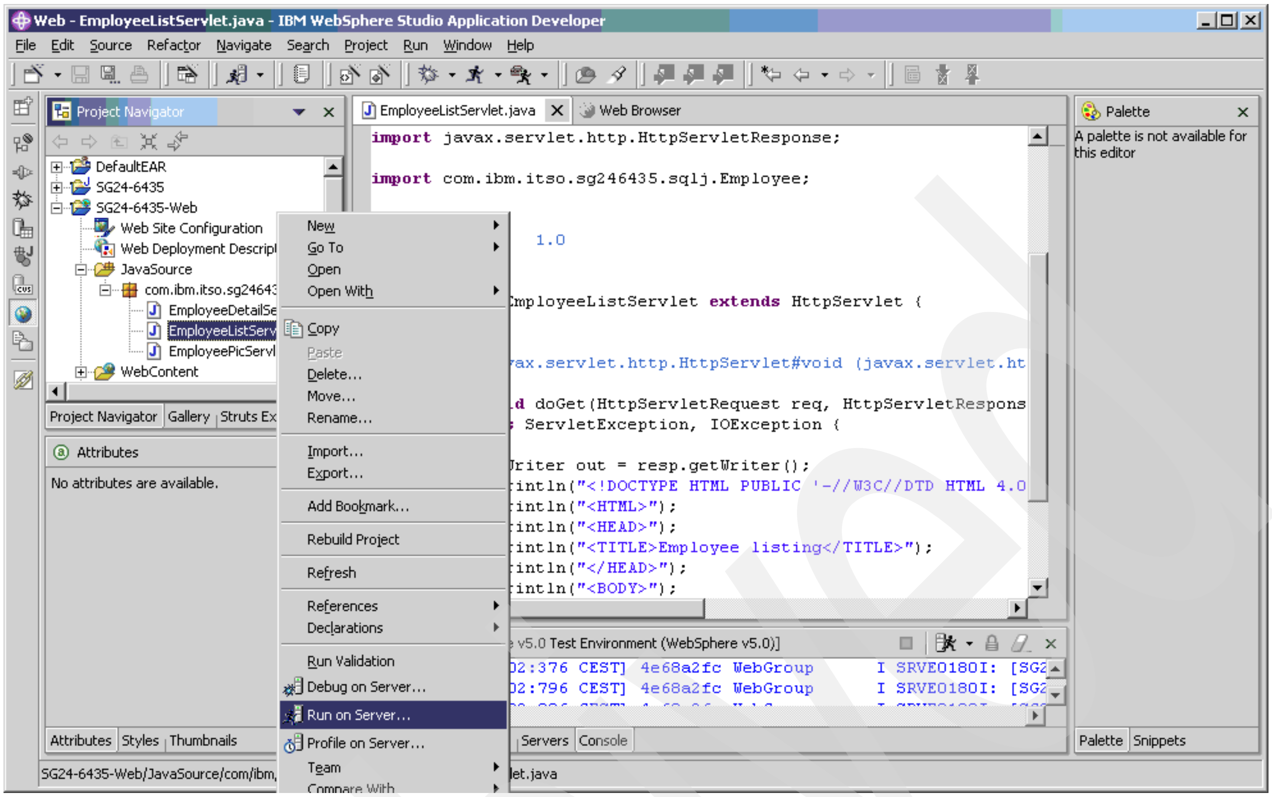

Figure 15-3 Running a Servlet in the WebSphere test environment

The following dialog prompts you for the TCP/IP port number the server will listen on. We stick with the default port number of 9080. Click **Finish**. WSAD now creates and starts the test environment and deploys our code. Be patient, this may take a while.

When the server has started up, WSAD opens a new editor window containing an embedded Web browser and requests the URL by which our Servlet can be invoked: `http://localhost:9080/SG24-6435/EmployeeListServlet`. The WebSphere test environment passes the request to the Servlet, which sends HTML text back to the client.

> **Note:** If something goes wrong, and you have to modify the Servlet code, you can run it anytime again as described above. However, we noticed that sometimes the embedded browser window does not reload the page, even if you press the Refresh button. If this happens, simply close the browser window and try again. You can also use an external browser to test your application, and just copy and paste the URL into the browser's address bar.

The Web browser window now displays the heading line (the text between the <h1> and </h1> tags). We are now ready for the interesting part.

15.2.3 Displaying the employee list

Now that the skeleton of our Servlet is ready, we can go for the interesting part—displaying the list of employees on record. A natural way of presenting the list is a tabular format; in HTML, what you do is create a table.

An HTML table, in its simplest form, looks like Example 15-2 on page 254.

Example 15-2 HTML table

```
<table>
  <tr>
    <td>Cell data for first in row</td>
    <td>Cell data for second in row</td>
    ... more cell data ...
  </tr>
  <tr>
    ... second row ...
  </tr>
  ... more rows ...
</table>
```

To display the employee list, we show the data for each employee in a row, with the individual attributes being the columns.

As mentioned before, we make use of the Employee class developed in Chapter 11, “SQLJ revisited” on page 199. To be able to do this, we have to tell WSAD that code in the SG24-6435 project should be available to the SG24-6435-Web project; that is, to include SG24-6435 in the Java build path. Also, we have to make sure that the binary code sitting in SG24-6435 is packaged with SG24-6435-Web when deploying the application.

To set the Java build path, and to include the binary code, perform the following steps:

1. Select **SG24-6435-Web** in the Navigator view and select **Properties** from the context menu.
2. Select **Java Build Path** and switch to the **Projects** pane. Check SG24-6435.
3. Then select the **Libraries** tab, click **Add External JARs...** and add the sqlj.zip file (that lives in ..\SQLLIB\Java). Otherwise you may get messages indicating that `The project was not built since the classpath is incomplete. Cannot find the class file for sqlj.runtime.ref.ResultSetIterImpl.`
4. Select **Web Library Projects** then click **Add**.
5. In the dialog box that opens, click **Browse**. Select the **SG24-6435** project then click **OK**.
6. WSAD automatically suggests a name of SG24-6435.jar. This is the name of the Jar file that will be packaged together with the Web application, containing the binary code in the SG24-6435 project. Click **OK**.
7. Click **OK** to end the Project Properties dialog.

Now we are ready to use the Employee class from our Servlet. We start by implementing a simple utility method that embeds a value passed in as argument in HTML `<td>` and `</td>` tags (Example 15-3).

Example 15-3 The EmployeeListServlet.printCol() utility method

```
/**
 * Print the supplied value to the servlet output stream
 * embedded in a <code>td</code> element.
 *
 * @param out
 *   The servlet output stream.
 * @param value
 *   The value to be printed.
 */
private void printCol(PrintWriter out, String value) {
  out.println("<td>");
```

```
  out.println(value);
  out.println("</td>");
}
```

The next step is to write a method that generates one table row, representing an employee.

Add an import statement for the Employee class:

```
import com.ibm.itso.sg246435.sqlj.Employee;
```

Then add the `printRow()` method (Example 15-4). This method makes use of the utility method above, making the code easier to read and maintain.

Example 15-4 The EmployeeListServlet.printRow() method

```
/**
 * Print employee data to the servlet output stream
 * in an HTML table row.
 *
 * @param out
 *   The servlet output stream.
 * @param emp
 *   The employee to be displayed.
 */
private void printRow(PrintWriter out, Employee emp) {
  out.println("<tr>");
  printCol(out, emp.getEmpNo());
  printCol(out, emp.getLastName());
  printCol(out, emp.getFirstName() + ' ' + emp.getMiddleInitial());
  out.println("</tr>");
}
```

Finally, we implement a method `printTable()`, which iterates through all employees, calling the `printRow()` method in turn for each employee.

Example 15-5 The EmployeeListServlet.printTable() method

```
   private static Employee.Ctx ctx;

/**
* Prints the table of employee data.
* *
* @param out
* The servlet output stream.
*
* @exception SQLException
* A database error occurred.
*/
private void printTable(PrintWriter out) throws SQLException {
   out.println("<table>");
   ctx = new Employee.Ctx();
   Employee.EmployeeIterator iter = Employee.findAll(ctx);
   try {
      Employee emp;
      while ((emp = Employee.fetch(iter)) != null)
         printRow(out, emp);
      out.println("</table>");
   } finally {
      iter.close();
```

```
    }
}
```

Do not forget to add the import statement for `java.sql.SQLException` to the program.

The last thing to add is the invocation of the `printTable()` method to the Servlet's `doGet()` method. This is shown in Example 15-6.

Example 15-6 Adapted doGet() method

```
public void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, IOException
{
  PrintWriter out = resp.getWriter();
  out.println("<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>");
  out.println("<HTML>");
  out.println("<HEAD>");
  out.println("<TITLE>Employee listing</TITLE>");
  out.println("</HEAD>");
  out.println("<BODY>");
  out.println("<H1>Employee listing</H1>");
  try {
    printTable(out);
  } catch (SQLException e) {
    throw new ServletException(e.getMessage(),e);
  }
  out.println("</BODY>");
  out.println("</HTML>");
  out.close();
}
```

15.3 Running the completed EmployeeList Servlet

By now, we are ready to run the complete EmployeeListServlet Servlet again. Right-click our Servlet and select **Run on Server…**. WSAD will prompt you for the server on which you want to run the Servlet; select the WebSphere Test Environment server created in "Testing the Servlet" on page 252. WSAD will display the embedded browser window again and invoke the Servlet.

> **Important:** When the browser window is already open, you have to press the Refresh button ( ) on this window in order to force a refresh.

Normally, the browser will now display a (not very helpful) error message, namely:

```
Error 500: Failed to load target servlet [EmployeeListServlet]
```

By examining the WebSphere log in the Console view, we find the actual reason for the error:

```
[7/25/03 22:06:01:871 CEST] 350b62f8 WebGroup      E SRVE0026E: [Servlet
Error]-[EmployeeListServlet]: java.lang.NoClassDefFoundError:
sqlj/runtime/ref/ResultSetIterImpl
   at java.lang.Class.newInstance0(Native Method)
   at java.lang.Class.newInstance(Class.java:262)
   at java.beans.Beans.instantiate(Beans.java:233)
   at java.beans.Beans.instantiate(Beans.java:77)
```

```
    at
com.ibm.ws.webcontainer.webapp.WebAppServletManager.loadServlet(WebAppServletManager.jav
a:188)
...
```

This indicates that the SQLJ runtime library is not on the server's classpath. To fix this, do the following:

1. Switch to the Servers view (select the **Servers** tab in the bottom window).
2. Double-click the WebSphere v5.0 Test Environment entry. An editor opens that allows you to modify the server's configuration.
3. In that editor, select the **Environment** tab then press **Add External JARs...**.
4. Navigate to your DB2 installation directory and select db2jcc.jar and the license file, **db2jcc_license_cisuz.jar**. Press **Open**.
5. Save the server configuration (select **File -> Save** or press Ctrl+S).
6. Since its configuration has changed, the server should be restarted as indicated in the Servers view (see Figure 15-4). Do so by selecting **Restart** from the context menu. The server shuts down, then restarts.

> **Tip:** The browser window in WSAD for Windows is an embedded Internet Explorer. Depending on your Internet Explorer configuration, you either see a generic, built-in error message, or the original error message from the Web server. To display the server error message, open Internet Explorer outside WSAD, and select **Tools -> Internet Options**. Switch to the **Advanced** tab and uncheck **Show friendly HTTP error messages**. (We feel the friendly message is rather unfriendly, at least if you are a Web developer).

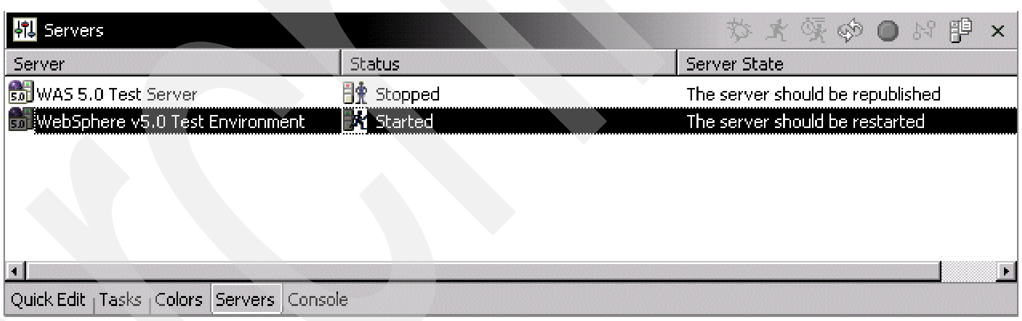

Figure 15-4 Server view indicating that the server should be restarted

Run the Servlet one more time by clicking the **Refresh** button in the Web browser.

This time, we receive another error message, indicating that no default context has been set up for SQLJ. The browser window displays:

```
Error 500: Found null DefaultContext...
```

Again, you can see the complete error message and stack trace by inspecting the WebSphere log in the Console view (Example 15-7).

Example 15-7 Sample error message from console

```
[10/26/03 23:12:59:852 PST] 3228927e WebGroup      I SRVE0180I: [SG24-6435-Web]
[/SG24-6435] [Servlet.LOG]: EmployeeListServlet: init
[10/26/03 23:13:00:784 PST] 3228927e WebGroup      E SRVE0026E: [Servlet Error]-[found
        null connection context]: java.sql.SQLException: found null connection context
    at sqlj.runtime.error.Errors.raiseError(Errors.java:125)
```

```
at sqlj.runtime.error.Errors.raiseError(Errors.java:73)
at
   sqlj.runtime.error.RuntimeRefErrors.raise_NULL_CONN_CTX(RuntimeRefErrors.java:136)
at com.ibm.itso.sg246435.sqlj.Employee.findAll(Employee.sqlj:223)
at
com.ibm.itso.sg246435.web.EmployeeListServlet.printTable(EmployeeListServlet.java:91)
at com.ibm.itso.sg246435.web.EmployeeListServlet.doGet(EmployeeListServlet.java:38)
.....
```

To solve this problem, we can set up a data source in the WebSphere configuration by the name of jdbc/empdb, as we have an explicit context declaration with that name in the Employee class.

Setting up a data source in WebSphere Studio

To set up a data source in WebSphere Studio, perform the following steps:

1. Reopen the WebSphere v5.0 Test Environment in the Servers view.
2. In the editor for the server configuration, select the **Data Source** tab. From the JDBC provider list, click the **Add** button to the right of the JDBC provider list (Figure 15-5). At the time of writing, WSAD did not support the new Universal Driver for DB2 for z/OS, so you should not select Default DB2 JDBC Provider.

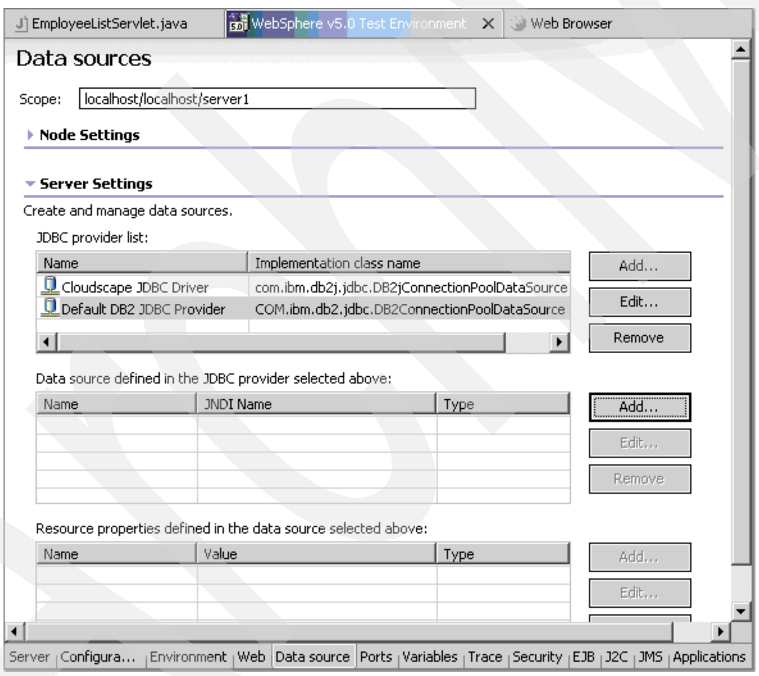

Figure 15-5 Creating a new data source

3. On the Create a JDBC Provider window, select **User-defined** as the database type, and **User-defined JDBC provider** as JDBC provider type (Figure 15-6 on page 259), and click **Next**.

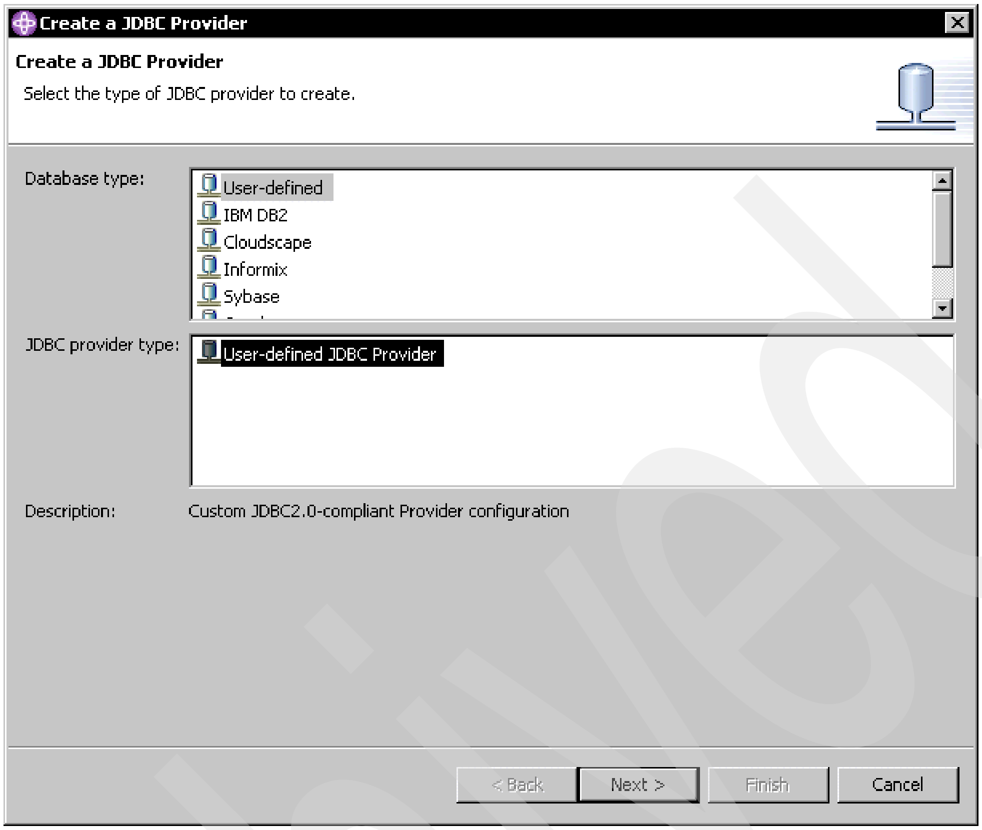

Figure 15-6 Create a JDBC Provider

4. Specify a name for the JDBC provider, for example, IBM Type 2 JDBC provider, as well as a description. Select **com.ibm.db2.jcc.DB2ConnectionPoolDataSource** as the implementation class name, and add the correct jar files to the class path using the **Add External Jars** button, as shown in Figure 15-7 on page 260, and click the **Finish** button.

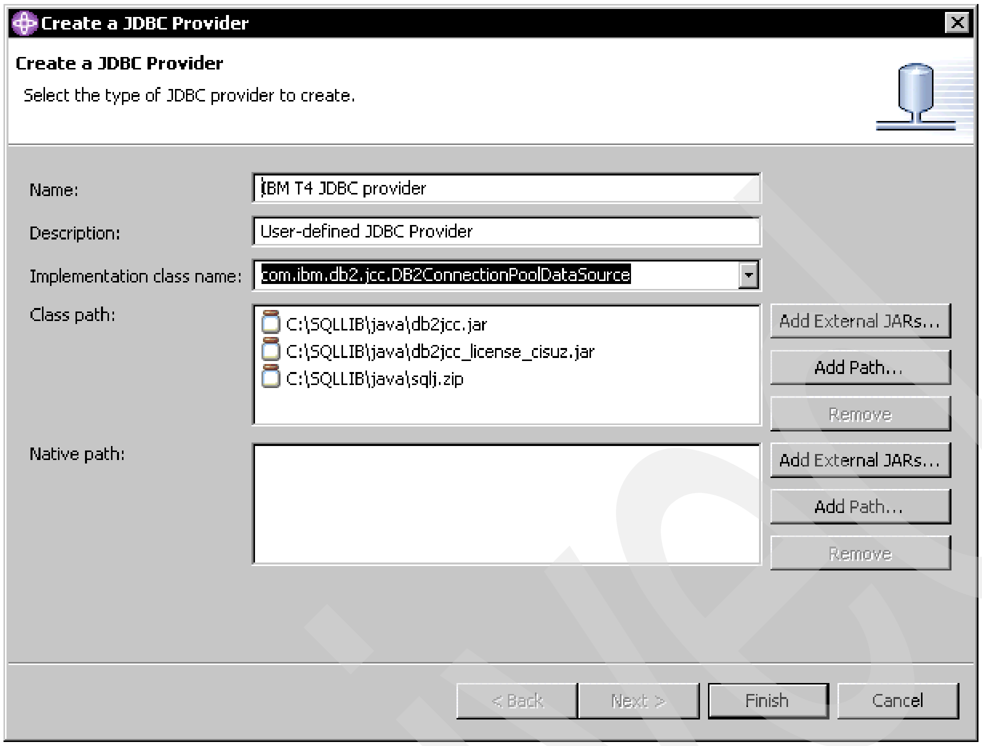

Figure 15-7 Create a JDBC Provider - 2

Now we are back to the window shown in Figure 15-5 on page 258, with your newly defined JDBC provider in the list. Now it is time to define the data source using the newly define JDBC provider. (Make sure the newly created JDBC provider is highlighted, which it normally is after you have created it.)

5. Click **Add** beside the (currently empty) data source list.
6. On the next dialog, select **DB2 Universal JDBC Driver Provider** and **Version 5.0 data source**. Press **Next**.
7. The next dialog prompts you to configure properties for the new data source. We only need to provide values for the required fields, Name and JNDI name. In the Name field, enter a descriptive name such as `Data source T4`. The most important property is the JNDI name; enter `jdbc/empdb` (Figure 15-8 on page 261). This is the data source name when setting up the connection context in the Employee class. Also specify `com.ibm.websphere.rsadapter.DB2DataStoreHelper` as the DataSource helper class name. Click **Next**.

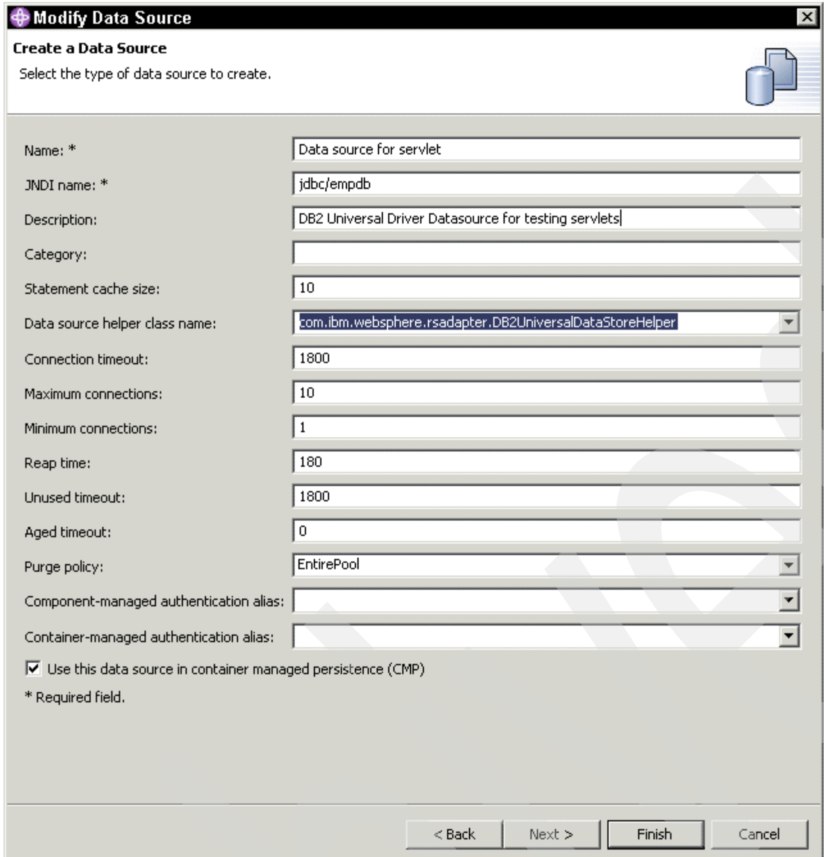

Figure 15-8 Create a data source

8. Now, we have to configure a few properties specific to DB2 data sources. The following dialog displays a list of configurable properties, which you can set in the Value field below the list. Select the properties **databaseName** (Figure 15-9 on page 262), **serverName**, and **portNumber**, in turn, and enter the corresponding values, namely, your DB2 subsystem name, your z/OS system's (IP) host name or IP number, and the DB2 port number. Click **Finish**.

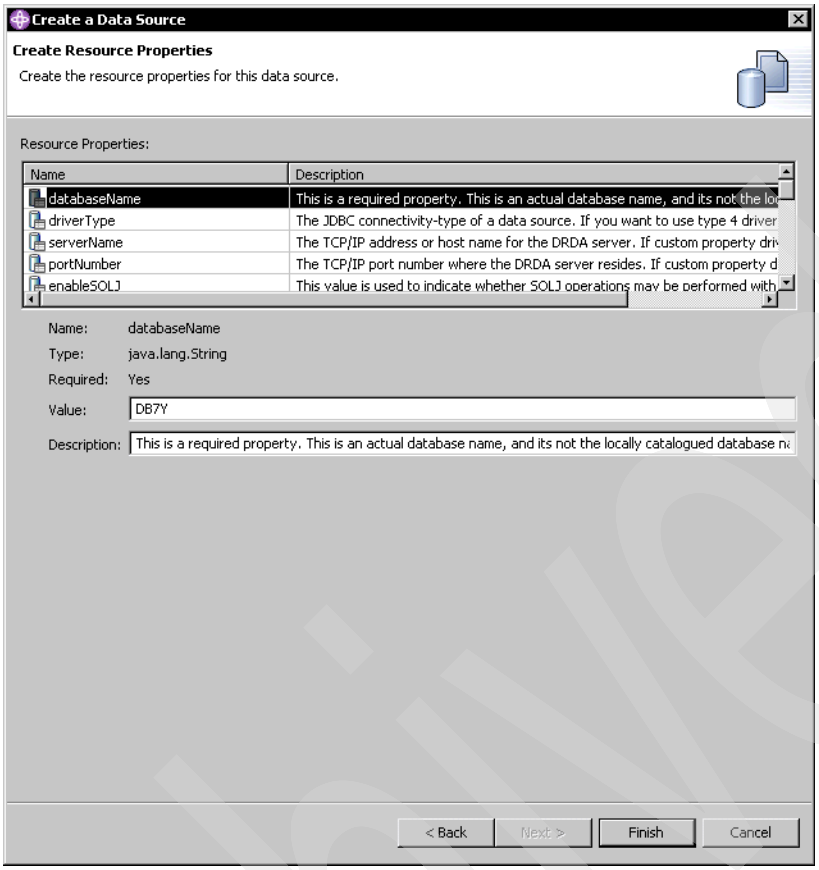

Figure 15-9 Create a Data Source -2

9. We are now back in the Data Source tab of the server configuration. The final configuration step is to add a default user and password for the data source. We decide to define it as a resource property. Beside the Resource properties list, click the **Add** button, and fill out the dialog as shown in Figure 15-10. Repeat this step for the password property.

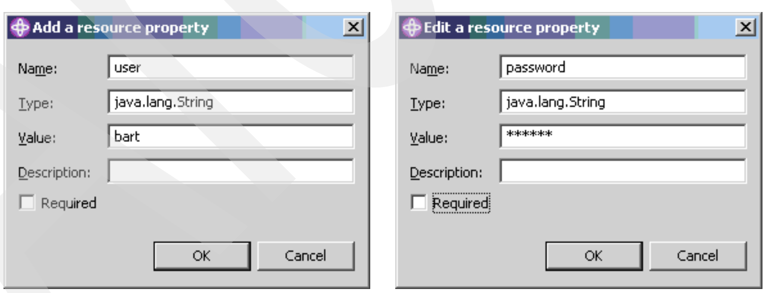

Figure 15-10 Add userName and password properties to the data source configuration

10. Save the server configuration (select **File -> Save** or press Ctrl+S).

11. Restart the server (see page 257).

Note: During our testing we found that WebSphere Studio would regularly mess up the server configuration definition, or would not pick up changes made to the configuration. This was without any doubt due to the fact that we were working with an early copy of the product, and those problems are not very likely to occur in your case. However, when they do, it is best start from scratch; that is, delete the complete server (Websphere V5.0 Test Environment), stop/start WSAD, and define the server as well as the data source from scratch.

Now we are ready to run the Servlet once more. Republish the server configuration and restart the server. If the browser window showing the error message is still open, you can simply click its **Refresh** button; otherwise, select the Servlet and click **Run on Server…**.

Now the SQLJ runtime will create connection context from the data source registered under `jdbc/empdb`, and the Servlet runs successfully. The output in the browser window should look similar to Figure 15-11.

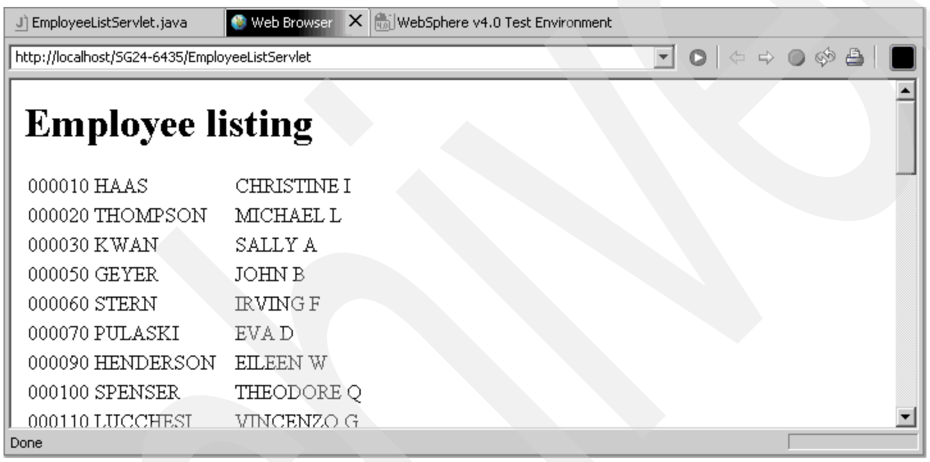

Figure 15-11 Output from running the EmployeeList Servlet

We did it!

15.4 Creating the EmployeeDetail Servlet

The EmployeeDetail Servlet displays detail information about an employee. To create the EmployeeDetail Servlet, follow the same steps as for the EmployeeList Servlet.

Example 15-8 shows the `doGet()` method and a helper method that prints one row of employee data. However, we have one problem here. How do we tell the Servlet which employee to display? For the time being, we hardcode an employee number (see note 1 below).

Example 15-8 EmployeeDetailServlet.doGet(), first version

```
package com.ibm.itso.sg246435.web;

import java.io.IOException;

import javax.servlet.Servlet;
import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


import com.ibm.itso.sg246435.sqlj.Employee;
import java.io.PrintWriter;

/**
 * @version 1.0
 * @author
 */
public class EmployeeDetailServlet extends HttpServlet implements Servlet {

   private static Employee.Ctx ctx;

   /**
   * @see javax.servlet.http.HttpServlet#void (javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
   */
   public void doGet(HttpServletRequest req, HttpServletResponse resp)
   throws ServletException, IOException {
   PrintWriter out = resp.getWriter();
   String empno = "000130";                                                    1
   out.println("<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>");
   out.println("<html>");
   out.println("<head><title>Employee detail</title></head>");
   out.println("<body>");
   out.println("<table>");
   try {
   ctx = new Employee.Ctx();
   Employee emp = Employee.findByPrimaryKey(ctx,empno);                        2
   printRow(out, "Serial", emp.getEmpNo());                                    3
   printRow(out, "Name", emp.getLastName() + ", " + emp.getFirstName());
   printRow(out, "Hired", emp.getHireDate());
   printRow(out, "Salary", emp.getSalary());
   } catch (Exception e) {
   throw new ServletException(e);
   }
   out.println("</table>");
   out.println("</body>");
   out.println("</html>");
   }
   /**
   * Prints one row of employee information to the
   * servlet output stream. The left column shows a
   * label, the right column the actual data.
   *
   * @param out
   * The servlet response output stream.
   * @param label
   * The label to be displayed in the left column.
   * @param value
   * The value to be displayed in the right column.
   */
   private void printRow(PrintWriter out, String label, Object value) throws IOException {
   out.println("<tr>");
   out.println("<td>" + label + ":</td>");
   out.println("<td>" + value + "</td>");
   out.println("</tr>");
   }
```

```
}
```

Notes on Example 15-8 on page 263:

1. Employee number hardcoded—we fix this later.
2. Retrieve employee data.
3. Print one HTML table row for each attribute we want to display.

Do not forget to include the `import com.ibm.itso.sg246435.sqlj.Employee;` statement for the Employee class.

15.5 Creating the EmployeePic Servlet

Finally, we create a Servlet to display the employee's picture. As in "Creating the EmployeeDetail Servlet" on page 263, we hardcode the employee number for the time being.

Again, create and test the Servlet as described in "Creating the EmployeeList Servlet" on page 251. The `EmployeePicServlet.doGet()` method is shown in Example 15-9.

Example 15-9 EmployeePicServlet.doGet(), first version

```
package com.ibm.itso.sg246435.web;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import javax.servlet.Servlet;
import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ibm.itso.sg246435.sqlj.Employee;

/**
 * @version 1.0
 * @author
 */
public class EmployeePicServlet extends HttpServlet implements Servlet {

   /**
   * @see javax.servlet.http.HttpServlet#void (javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
   */
   private static Employee.Ctx ctx;

   public void doGet(HttpServletRequest req, HttpServletResponse resp)
   throws ServletException, IOException {
   resp.setContentType("image/bmp");                                              1

   OutputStream out = null;
   InputStream pic = null;
   String empno = "000130";                                                      2
   try {
```

```
   ctx = new Employee.Ctx();
   Employee emp = Employee.findByPrimaryKey(ctx,empno);
   out = resp.getOutputStream();                                             3
   pic = emp.getPicture(ctx);
   copy(pic, out);
   } catch (Exception e) {
   throw new ServletException(e);
   } finally {
// Clean up                                                                  4
   if (pic != null) pic.close();
   if (out != null) out.close();
   }
   }
   private void copy(InputStream in, OutputStream out) throws IOException {  5
   byte[] buf = new byte[1024];
   int nread;
   while ((nread = in.read(buf)) > 0)
   out.write(buf, 0, nread);
   }

}
```

Notes on Example 15-9 on page 265:

1. This tells the client (usually a Web browser) the content type of the response. The default for HttpServlets is a content type of “text/html”, indicating that the response is an HTML document. Since we return a different content type (a GIF image), we have to override that default.
2. Again, for the time being the employee number is hardcoded.
3. While the previous examples used the getWriter() method to obtain the Servlet output stream, this Servlet uses getOutputStream(). The getWriter() method is used when your content type is text (such as HTML), whereas getOutputStream() must be used for binary content types.
4. To make sure that resources are freed even in the case of an error, the cleanup code is in a finally block.
5. This is a utility method that reads the input stream in chunks and copies the data to the output stream until no more data is available.

15.6 Putting it together

In this section we combine the Servlets developed in the previous sections.

First, we have to find a way to pass parameters to a Servlet and to retrieve the parameter values in the Servlet code. To solve the first problem, there are basically two mechanisms by which a Servlet (or any other component creating dynamic Web content, for that matter) can receive parameters, namely the GET method and the POST method. The former works by extending the URLs, while the latter works on the HTTP protocol level. We will not discuss the POST method here since it is primarily used for form input.

To pass a parameter to a Servlet via the GET method, you append the parameter names and parameter values to the URL after a preceding question mark (?). You separate individual parameters by an ampersand (&), and parameter names from parameter values with an equals sign (=).

15.6.1 Modifying the EmployeeList Servlet

First, we modify the employee list Servlet so that it displays the individual entries not as plain text, but as a hyperlink that can take us to the corresponding detail page.

Modify the `printRow()` method in EmployeeListServlet as shown in Example 15-10.

Example 15-10 EmployeeListServlet.printRow() modified

```
private void printRow(PrintWriter out, Employee emp) {
  out.println("<tr>");
  printCol(out, "<a href='EmployeeDetailServlet?empno=" + emp.getEmpNo() + "'>" +
    emp.getEmpNo() + "</a>");
  printCol(out, emp.getLastName());
  printCol(out, emp.getFirstName() + ' ' + emp.getMiddleInitial());
  out.println("</tr>");
}
```

Now run the Servlet again. The employee number will now appear as a hyperlink. When hovering over the hyperlink, you will see in your browser's status bar that the link target is of the form:

```
http://localhost:9080/SG24-6435/EmployeeDetailServlet?empno=xxxxxx
```

Where *xxxxxx* is the employee number.

If you click the hyperlink, the EmployeeDetailServlet is invoked and passed an employee number. Of course, since the EmployeeDetailServlet does not use the parameter right now, following the link always takes you to the (hardcoded) employee number 000130.

15.6.2 Modifying the EmployeeDetail and EmployeePic Servlets

With the modification in the previous section, we are now able to pass in the employee number to the EmployeeDetail Servlet as a parameter. To retrieve the parameter value, a Servlet calls the `getParameter()` method of its request object (the first parameter to the `doGet()` method). The argument to `getParameter()` is a string, the parameter name.

Thus, the required modification to both EmployeeDetailServlet and EmployeePicServlet is very simple: Just replace, in their respective `doGet()` methods, the lines:

```
String empno = "000130";
```

With:

```
String empno = req.getParameter("empno");
```

> **Note:** In a real application, you probably want to perform some error checking to ensure that the parameter is not missing and of the correct format.

Finally, for the finishing touch, we modify EmployeePicServlet to display a default image when no picture of the employee was found in the database. Modify the doGet() method, replacing:

```
copy(pic, out);
```

With:

```
if (pic == null) {
  getServletContext().getRequestDispatcher("/images/noimage.gif").forward(req, resp);
} else {
  copy(pic, out);
}
```

Create a new folder called images under the Web Content folder, and put the noimage.gif file in there.

When you run the EmployeeList Servlet one more time, the links should take you to the correct detail page for each employee.

15.6.3 Using EmployeePicServlet from EmployeeDetailServlet

As mentioned before, we want to show the employee's photo from the detail page if one is available.

To do so, we simply add one row to the table containing the employee detail information. Unlike the other table rows, this one does not contain text, but an image. Modify the `doGet()` method of `EmployeeDetailServlet`, as shown in boldface in Figure 15-11.

Example 15-11 EmployeeDetailServlet displaying a picture

```
try {
  Employee emp = Employee.findByPrimaryKey(empno);
  out.println("<tr><img src='EmployeePicServlet?empno=" + empno + "'></tr>");
  printRow(out, "Serial", emp.getEmpNo());
  printRow(out, "Name", emp.getLastName() + ", " + emp.getFirstName());
  printRow(out, "Hired", emp.getHireDate());
  printRow(out, "Salary", emp.getSalary());
} catch (Exception e) {
  throw new ServletException(e);
}
```

Go back to the employee list and click the link for employee #000130 (or any other employee for whom a picture has been supplied). The final version of the detail page should now look similar to Figure 15-12 on page 269.

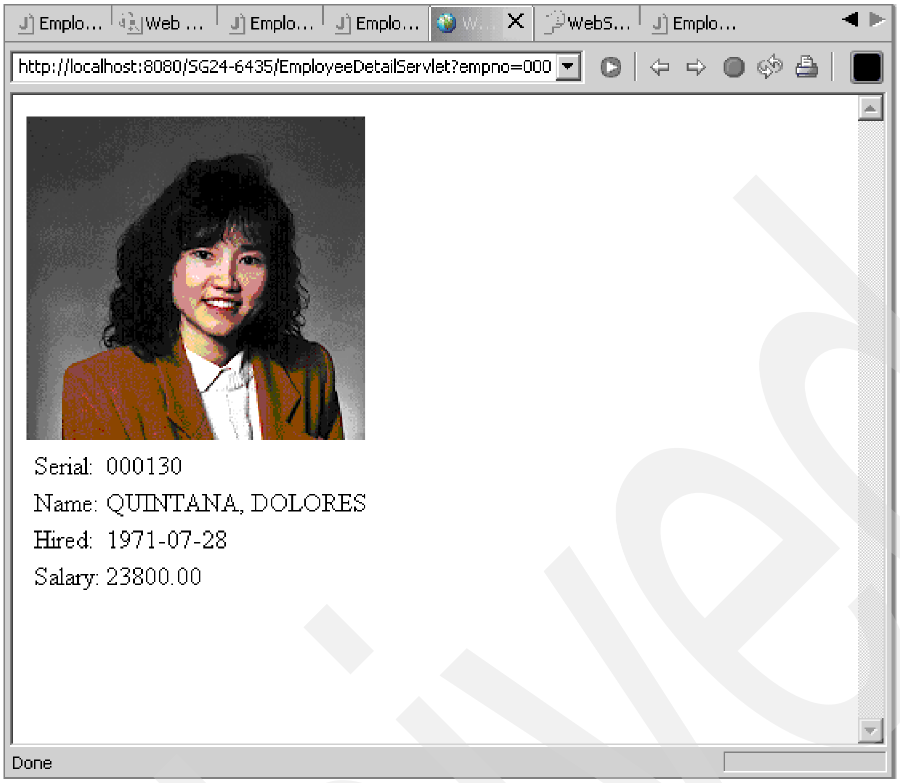

Figure 15-12 Output from EmployeeDetailServlet, final version

15.7 Deploying the application to WebSphere for z/OS

Until now, we used the WebSphere test environment of WSAD to develop and test our Servlets. Now it is time to deploy the application to a "real" Web application server.

15.7.1 Customizing the Web application to run as static SQL

However, first of all, we want to make sure that we are really running this application using static SQL to get the best performance. In WSAD, you simply have to generate and run your SQLJ Ant script again, as described in "Preparing SQLJ programs to use static SQL through WSAD" on page 157. Now we are ready to deploy the application on WAS for z/OS.

15.7.2 Creating a WAR file

To deploy an application to WebSphere (or another Web application server), you first package the application into a WAR (Web ARchive) file. This file contains all your code, static content, configuration information and other resources needed by the application at runtime. Then, you use the Web Application Server's administrative tools to make the application known to the server.

To create the WAR file from WSAD, do the following:

1. Select the Web project, **SG24-6435-Web**.
2. Select **File -> Export -> WAR file**.
3. In the Destination field, enter `S:\Bart\SG246435-static.war`.

4. Click **Finish**.

The WAR file is now ready for deployment to WebSphere Application Server (WAS) using the WAS administrative console.

15.7.3 Installing a new application on WAS for z/OS

As mentioned earlier, you use the WAS administrative console to manage your WeSphere for z/OS environment.

Log onto the administrative console

The WAS admin console uses a Web interface. Open a browser session and type in the URL to connect to the admin console application. In our case the URL is:

```
http://wtsc63.itso.ibm.com:9080/admin/
```

A screen similar to Figure 15-13 should appear.

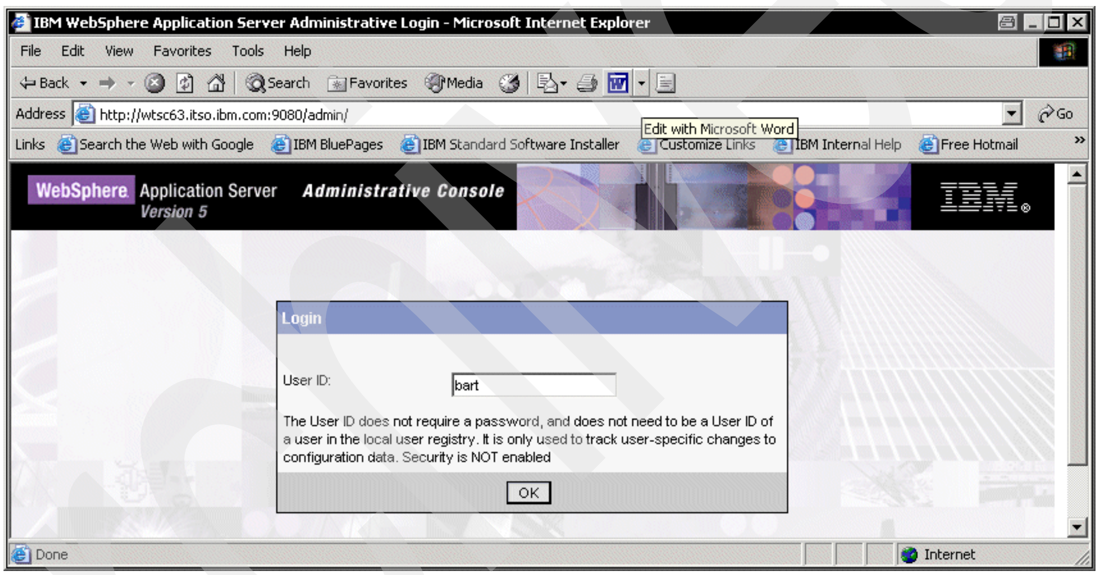

Figure 15-13 WAS administrative console

Type in your user ID and click **OK**.

Installing a new application

This takes you into the administrative console application. Expand the **Applications** tree option and click **Install New Application**. This takes you to a window similar to Figure 15-14 on page 271.

Select the **Local path** option and use the **Browse** button to find your war file (or just type in the complete file name). You also need to provide a context root to install your war file into. We use /itso for our application.

Click **Next** to continue.

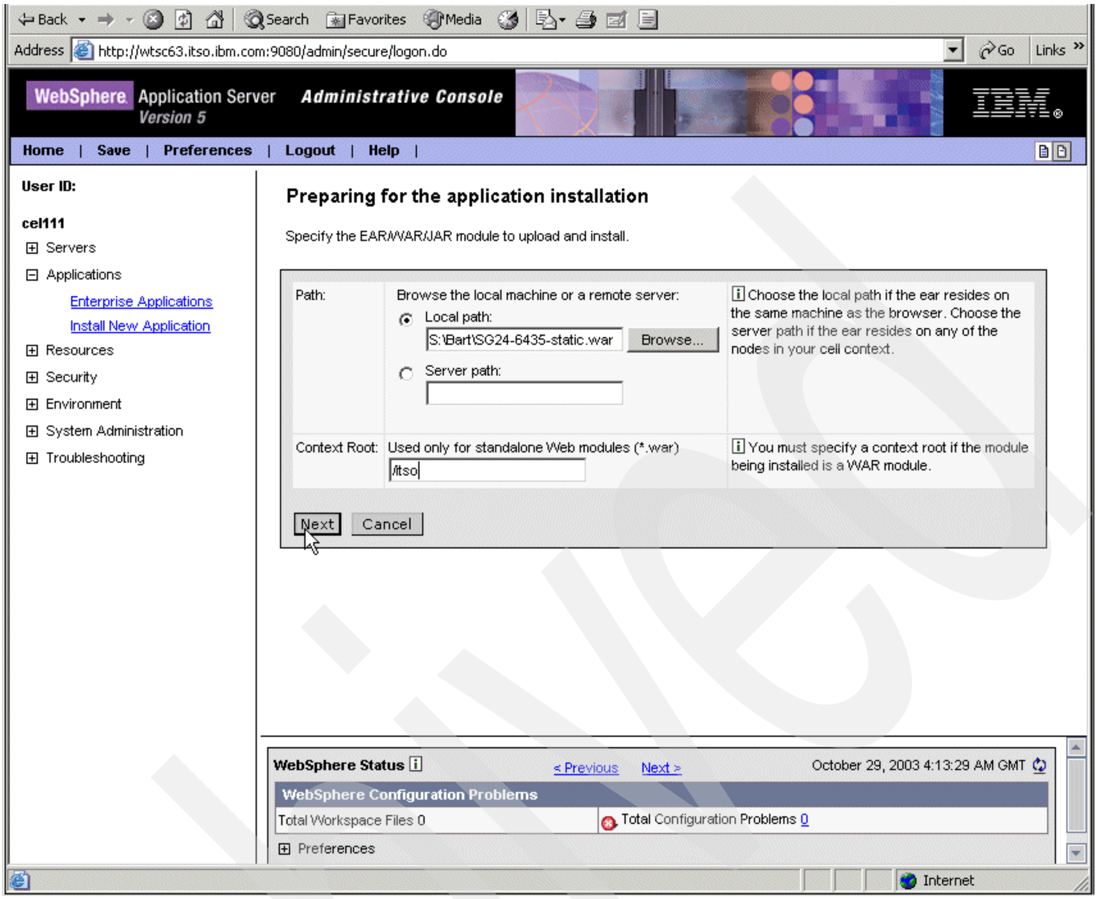

Figure 15-14 Install a new application

On the next window (Figure 15-15 on page 272) we do not need to change anything; just click **Next** to continue.

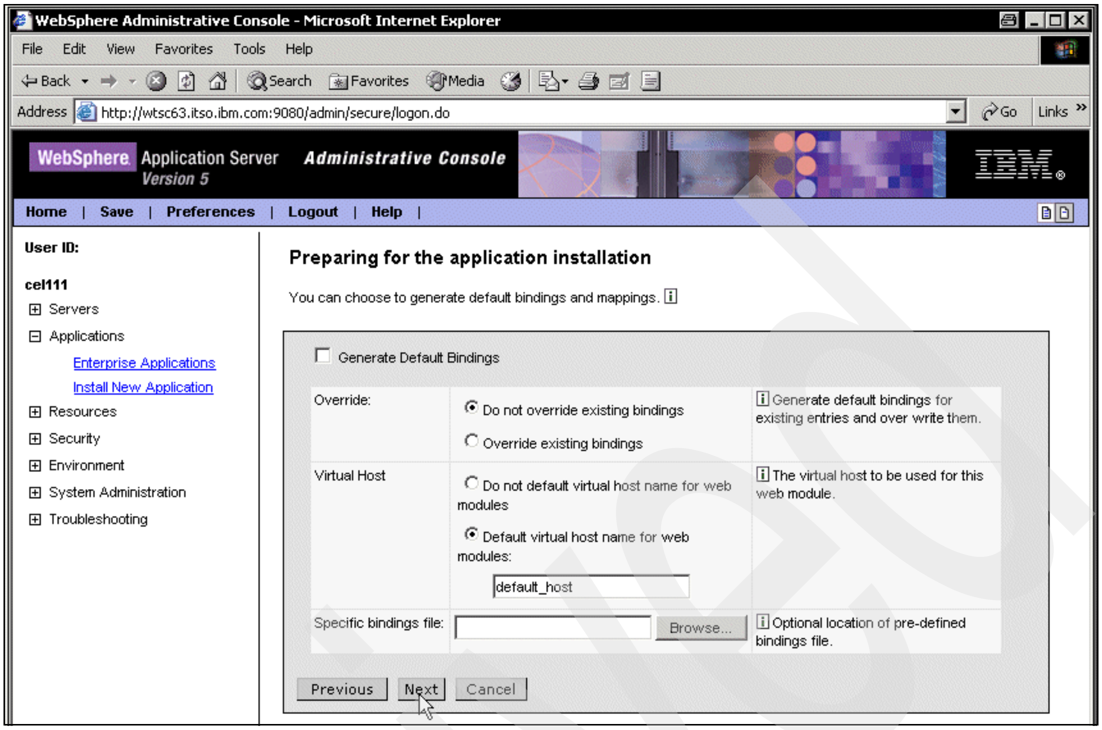

Figure 15-15 Install a new application - 2

On the next window (Figure 15-16 on page 273), all defaults are OK, so just click **Next**. In case you want to change the name of your application as it is known to WAS, you can change the application name here. The application name will show up later when we list the applications known to WAS, to activate our newly installed application.

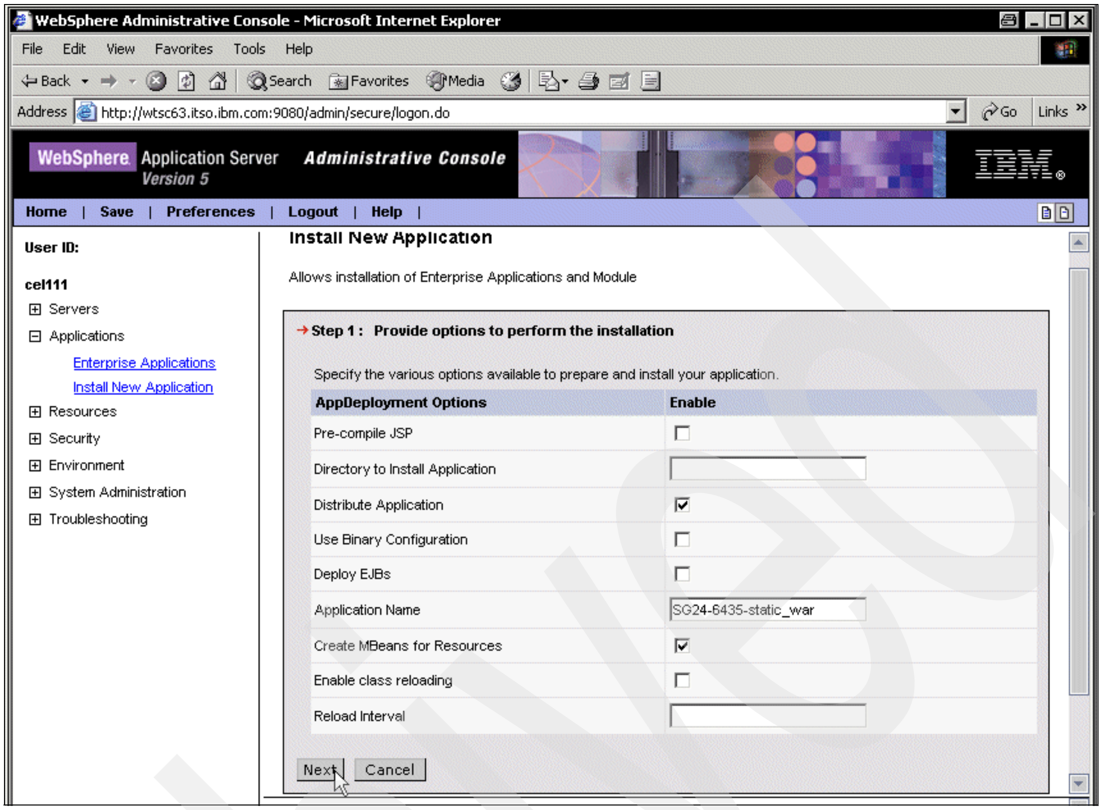

Figure 15-16 Install new a application - 3

Same thing for this window (Figure 15-17). Just accept the defaults by clicking **Next**.

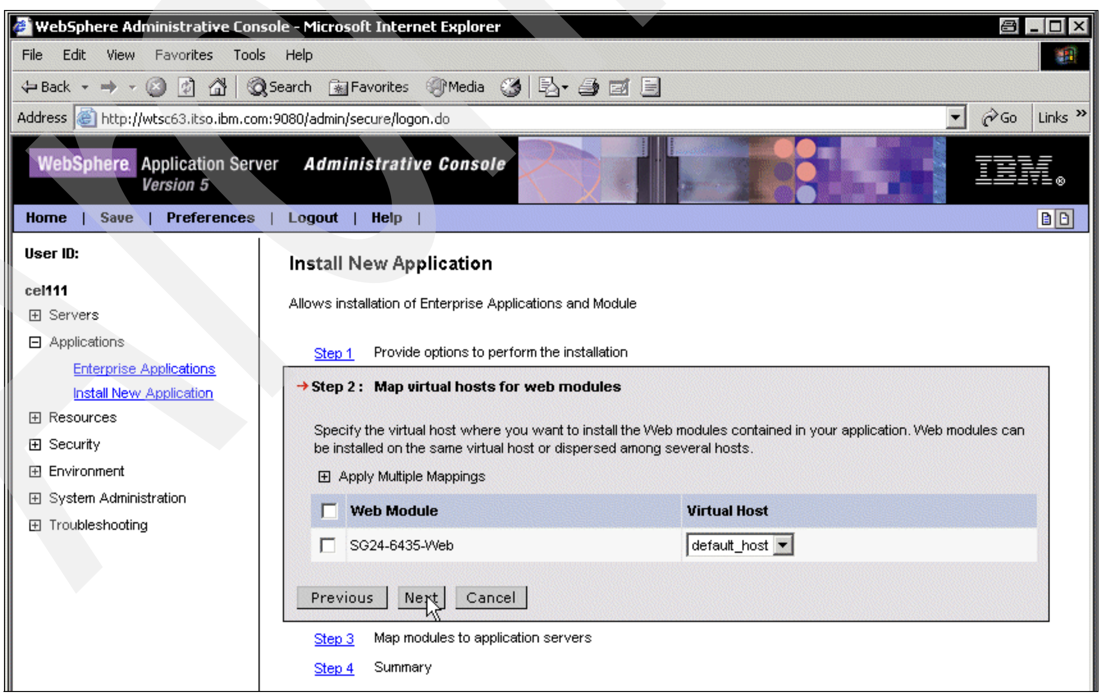

Figure 15-17 Install a new application - 4

On the next window (Figure 15-18), select the application server you want to intall on (`WebSphere:cell=cel111,node=nd111sc63,server=ws111sc63` in our case), and **check the box** next to SG24-6435, and click **Next**.

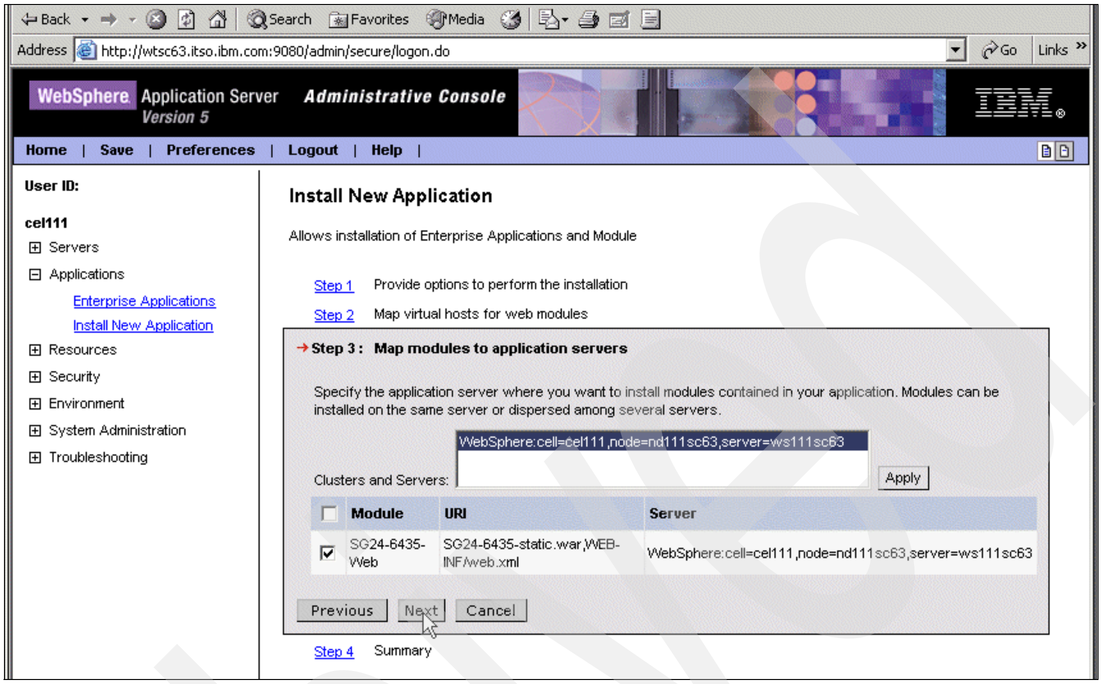

Figure 15-18 Install a new application - 5

The next window (Figure 15-19 on page 275) gives you an overview of the options you previously selected. Scroll to the bottom of the top window and click **Finish**.

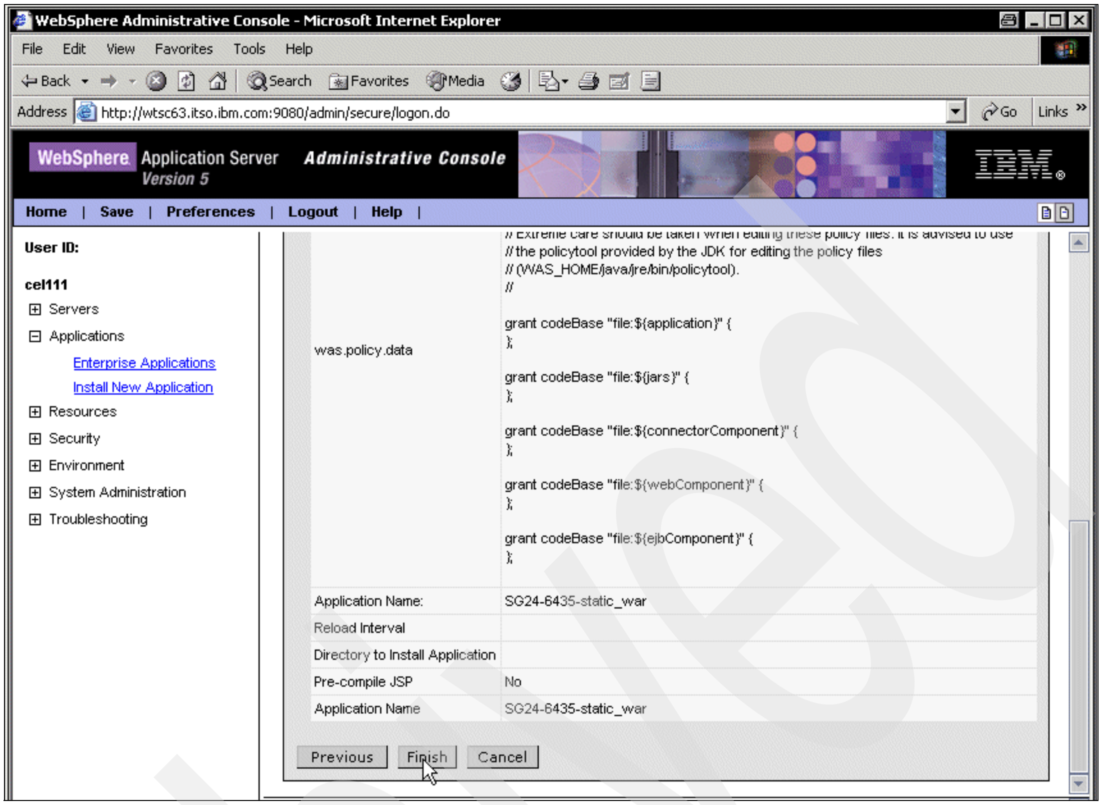

Figure 15-19 Install a new application - 6

Now it is time to save our changes. You do so by clicking **Save to Master Configuration**, as shown in Figure 15-20.

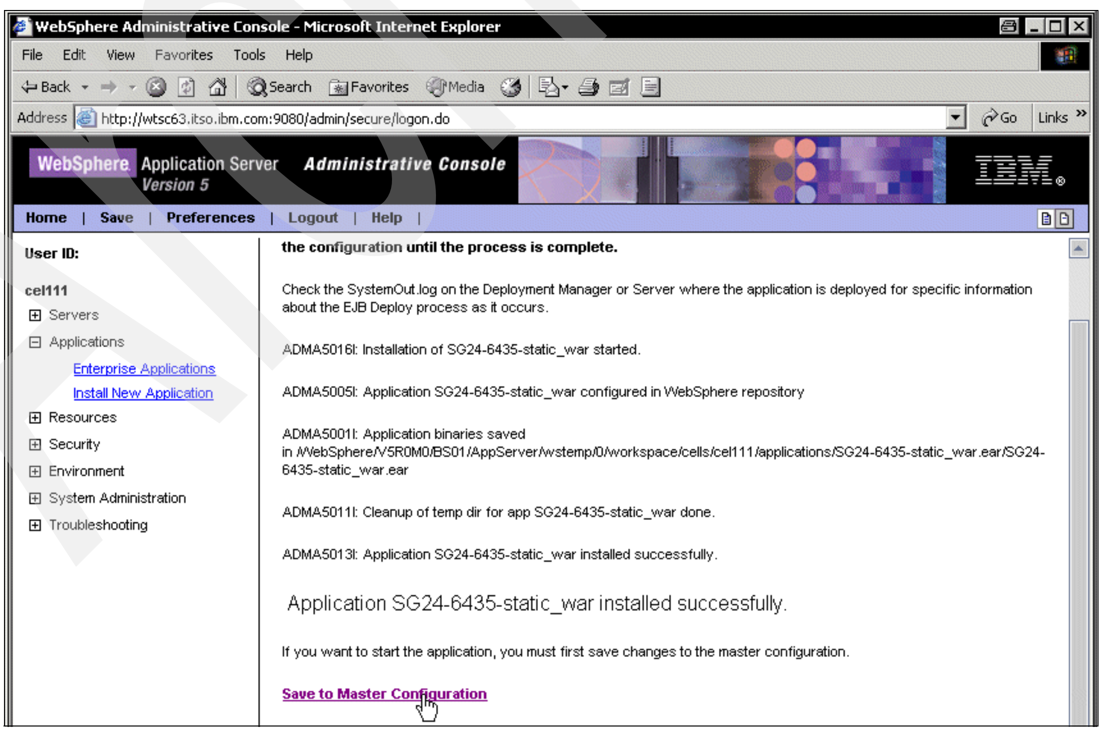

Figure 15-20 Save to master configuration

Click the **Save** button on the next window (Figure 15-21) to actually update the master repository.

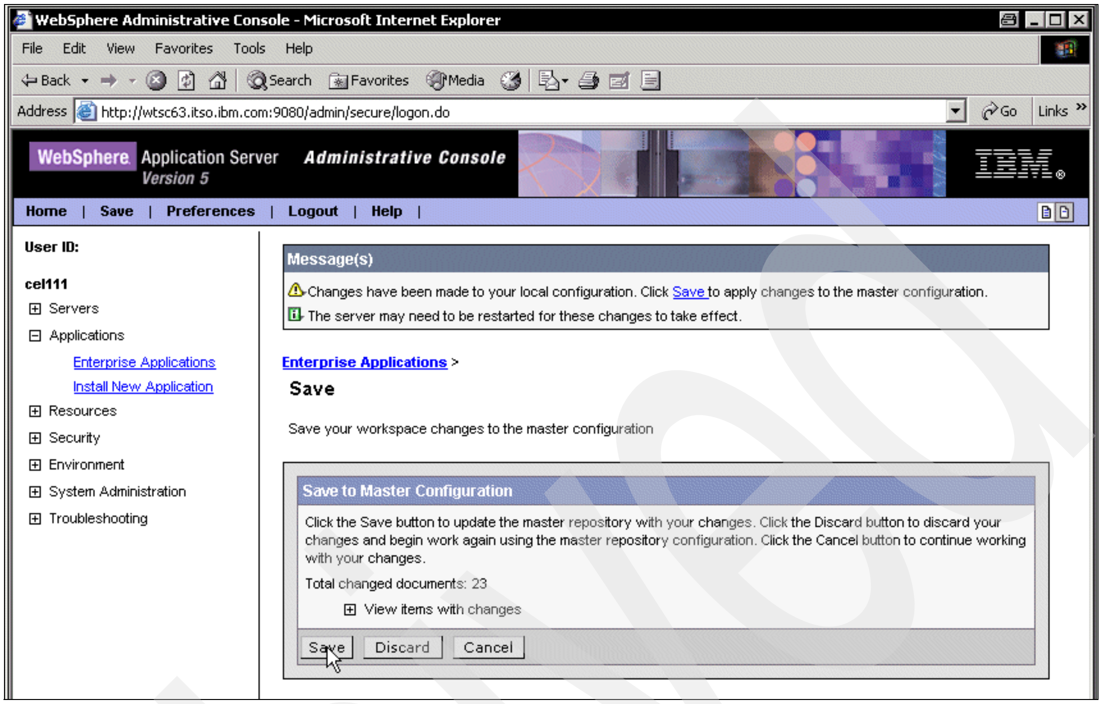

Figure 15-21 Save your changes to the master repository

After saving your changes, the admin console application takes you back to the main menu (Figure 15-22).

We are now ready to activate the application.

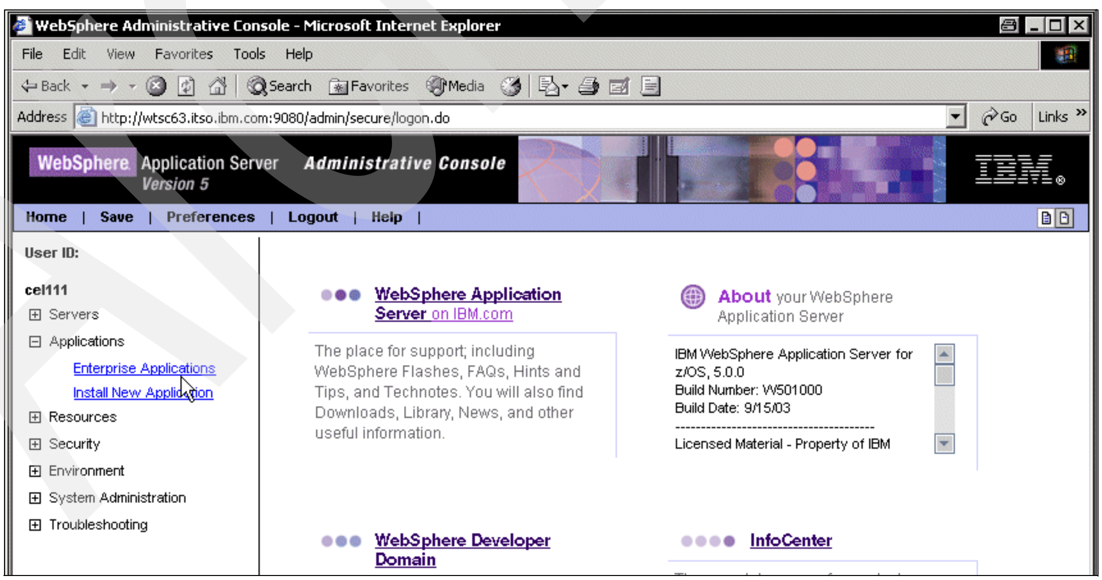

Figure 15-22 Administrative Console - Main menu

Starting an application

To activate an application, we select the **Enterprise Applications** option on the left pane. This takes us to the Enterprise Applications window (Figure 15-23). You can see that our newly installed application is stopped (indicated by the red X). To start the application, select the application by checking the box next to SG24-6435-static_war, and clicking the **Start** button.

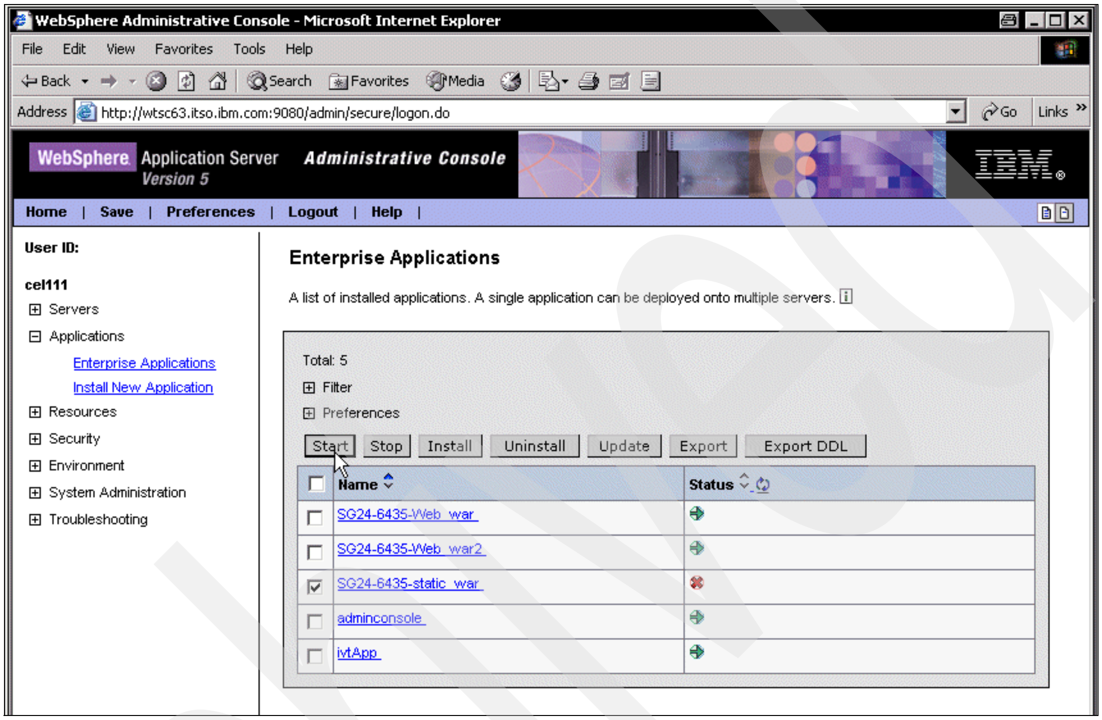

Figure 15-23 Start the application

If everything is OK, the application status turns into a green arrow, like the other applications. We are now ready to test.

15.7.4 Test the application

To test the application, open another browser window. Specify the following URL to invoke our Servlet:

```
http://wtsc63.itso.ibm.com:9080/itso/EmployeeListServlet
```

The result should be something similar to Figure 15-24 on page 278. Note that the context root (that we specified on the initial window when installing the new application) is part of the URL that you specify to invoke the Servlet.

Important: This example clearly shows the true portability of Java applications including the new JCC driver.

We developed and tested our application in WSAD on a workstation. We just exported the files and deployed them on a z/OS platform into a WebSphere Application Server, and we are up and running. We did not have to touch the application at all. The exact same code runs on WSAD, as inside our WAS on z/OS.

We even switched the driver type under the covers, without any changes to the application. Remember that when testing with WSAD, we used a T4 driver to connect to the DB2 for z/OS system (DB7Y). When running under the control of WAS for z/OS, we use a Type 2 connection to the same DB7Y system. Switching is transparant to the application, as it is done through the specification of the URL name that you use to set up the connection to your database. The specification of that URL is done in the datasource definition and is outside the application. The applation only uses a logical name, jdbc/empdb, to indicate the DB2 system it wants to access.

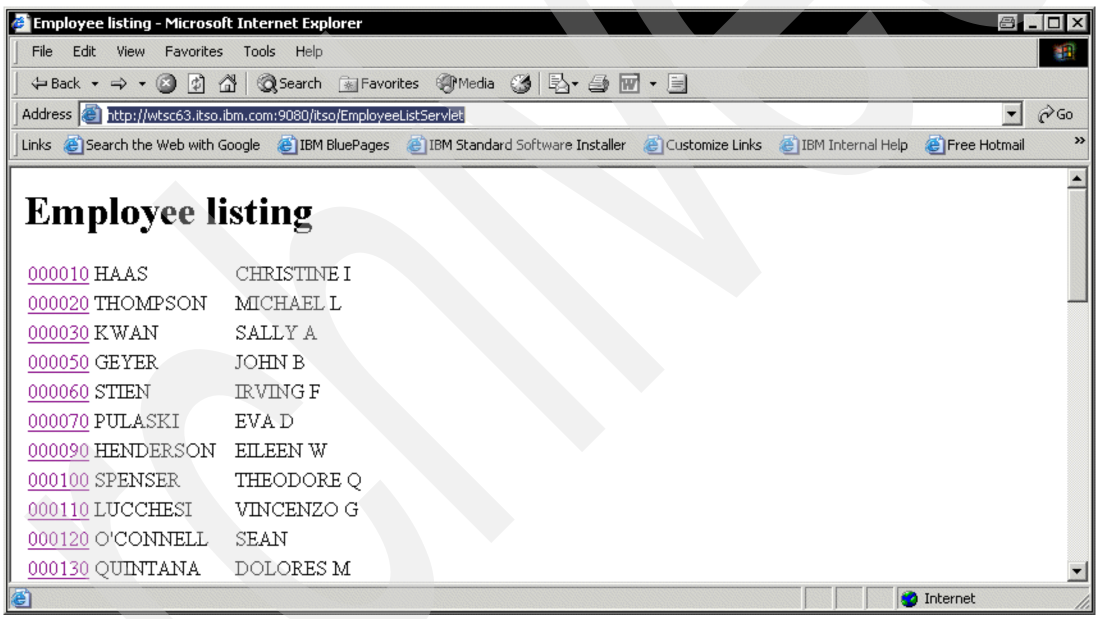

Figure 15-24 EmployeeListServlet result

16

JavaServer Pages

As a simpler alternative to Servlets, this chapter shows how to present DB2 data using JavaServer pages (JSPs).

While the JSP language per se does not have a built-in mechanism to access relational data, it can easily be extended by custom tag libraries. We demonstrate how to use a custom tag library for database access.

16.1 Introduction

In the previous chapter we used Servlets to access and display employee information. Although we reused the code developed in Chapter 11, “SQLJ revisited” on page 199, what we basically did was access the database using hand-written code, that is, we manually coded the logic for accessing the data, processing the query result sets, and generating the HTML output.

This made the code hard to read, and therefore hard to maintain. The HTML code to be produced is buried inside the Java code, and even with a very simple example like we developed, the HTML structure is difficult to understand.

In this chapter we take a different approach. We use JavaServer Pages (JSPs), which take the same approach as Servlets, but the other way round, so to speak. While our Servlets in the previous chapters were essentially Java code interspersed with HTML, JSPs are HTML code interspersed with Java code snippets. (JSPs are not restricted to generating HTML; it may just as well be XML, or any other content type supported by clients.)

> **Note:** In fact, JSPs are actually translated into Servlets by the application server behind the scenes.

One of the exciting features of JSPs is that the JSP language is designed to be extensible, using a feature called custom tag libraries. To use a custom tag library in your JSP, you declare the tag library in a so-called page directive (which is part of the JSP syntax). Once declared, you can use the tags provided by the library exactly like the built-in tags that are part of the JSP core language.

Although custom tag libraries are a relatively new feature of JSP, there are already quite a number of custom tag libraries around, which you can use for things like:

- Formatting strings, dates, and timestamps
- Internationalization
- Sending e-mail
- Processing XML documents
- Accessing databases

WSAD comes with several tag libraries that are ready to use. We will use a tag library for database access in this chapter, rewriting our employee list application to use JSPs.

> **Note:** WSAD comes with a powerful wizard that allows you to automatically create JSPs for database access. However, in this section we will code the JSPs manually in order to help you better understand what is going on.
>
> If you want to experiment with the Database Web Pages wizard, you can find it under **File -> New -> Web -> Database Web Pages**.

16.2 Creating the EmployeeList JSP

First of all, we have to tell WSAD that our Web application wants to use one of the tag libraries that come with the product.

1. Open the Properties page for the SG24-6435-Web project.
2. Select the **Web Project Features** category.
3. Check **Tag libraries for database access** (see Figure 16-1 on page 281).

4. Click the **Apply** button. This may take a while to complete. Then click **OK**.

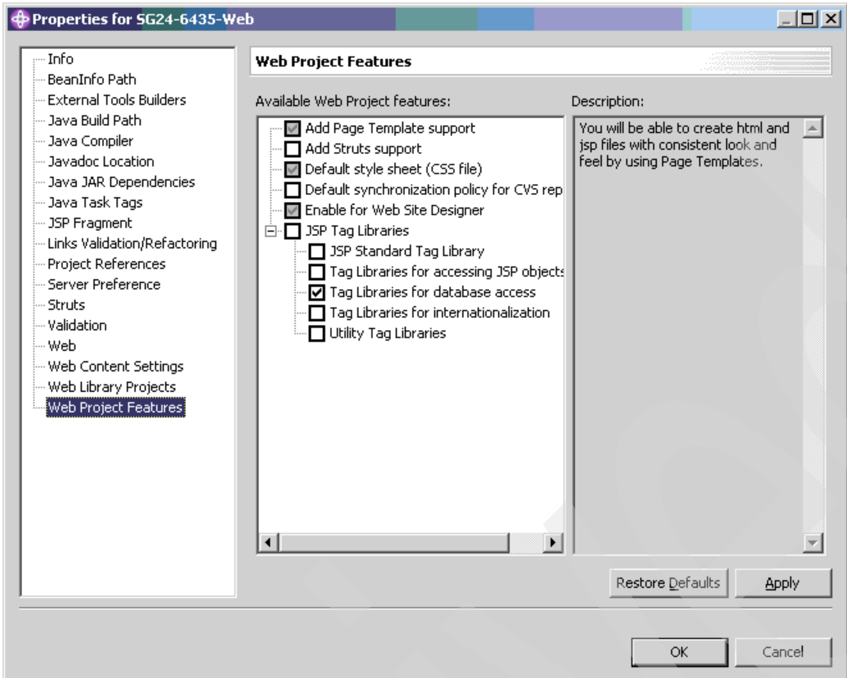

Figure 16-1 Including the database access JSP tag libraries

What happens is that WSAD copies the Jar files implementing the custom tags to your WebContent/WEB-INF/lib directory, ensuring that these Jar files are included with your application when you export it to a deployable format (that is, a .war or .ear file).

Next, we create the JSP skeleton by selecting **File -> New -> Other -> Web -> JSP file**. Click **Next**. The the New JSP File dialog appears (see Figure 16-2 on page 282).

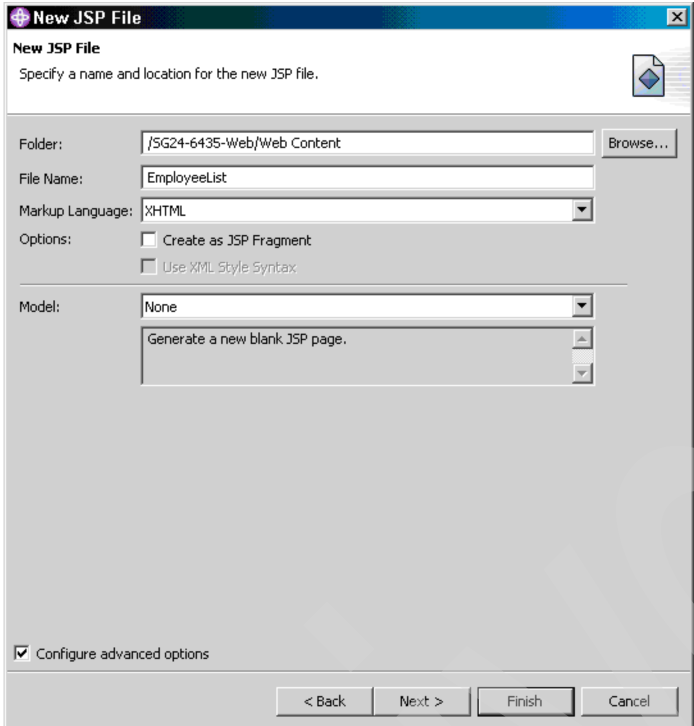

Figure 16-2 New JSP File wizard

Enter `EmployeeList` in the File Name text entry field. We choose XHTML as the markup language (HTML or Compact HTML would have been possible, too). Make sure that "Configure advanced options" is checked, and click **Next**.

In the next dialog window of the wizard, we tell WSAD that we want to include a custom tag library in the JSP. Click the **Add…** button, which will open the dialog in Figure 16-3 on page 283.

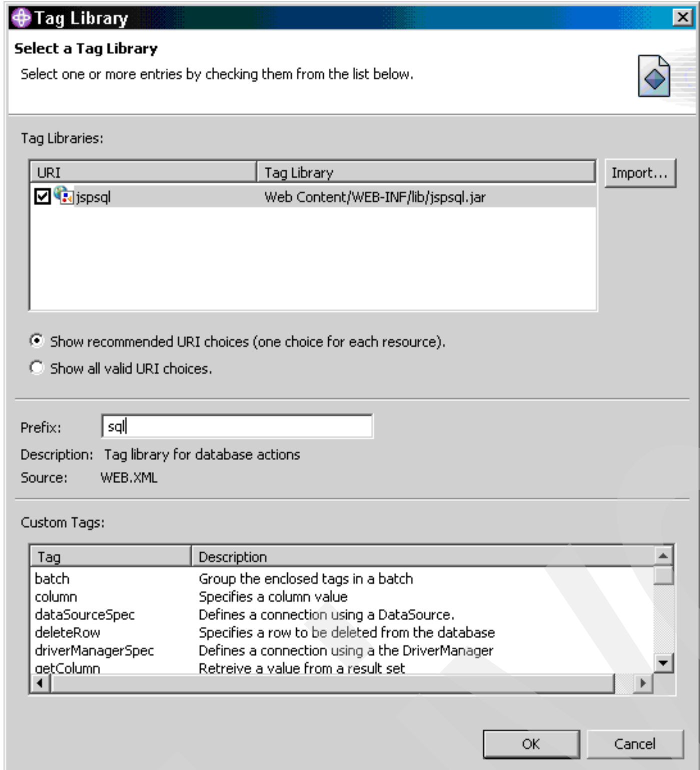

Figure 16-3 Select Tag Library

The Tag Libraries box lists all the tag libraries that are available in the project. Right now, it only lists the jspsql tag library that we included in the project at the beginning of this section. Check the library and enter `sql` in the Prefix text field. This tells the JSP engine that all custom tags defined by the library will be referenced in the JSP with a prefix of sql: (we could have used any other prefix as long as it does not collide with another prefix already in use). Click **OK**. We can leave the next dialog window as it is, and click **Next** to proceed to the **Method Stubs** panel. Here, make sure that "Add to web.xml" is checked, which causes a URL mapping to be set up for our JSP so it can later be invoked from a Web browser.

Now click **Finish**. WSAD creates a skeleton JSP file that reflects our settings from the New JSP Page wizard (Example 16-1). You may have to click the **Source** tab to see the actual JSP program skeleton.

Example 16-1 Skeleton JSP file generated by the New JSP Page wizard

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<%@ taglib uri="jspsql" prefix="sql" %>
<%@ page
language="java"
contentType="text/html; charset=ISO-8859-1"
%>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<meta name="GENERATOR" content="IBM WebSphere Studio" />
<meta http-equiv="Content-Style-Type" content="text/css" />
<link href="theme/Master.css" rel="stylesheet"
   type="text/css" />
```

```
<title>EmployeeList.jsp</title>
</head>
<body>
<p>Place content here.</p>
</body>
</html>
```

We are now ready to develop the JSP in more less the same way as the Employee List Servlet from Chapter 15, “Using Servlets to access DB2” on page 249. The fundamental difference is that we can focus on the HTML code, that is, on the presentation of the data, rather than on Java code, to generate HTML.

The full source code of EmployeeList.jsp is shown in Example 16-2.

Example 16-2 EmployeeList.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <%@ taglib uri="jspsql" prefix="sql" %>                                 1
    <%@ page
    language="java"
    contentType="text/html; charset=ISO-8859-1"
    %>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
    <meta name="GENERATOR" content="IBM WebSphere Studio" />
    <meta http-equiv="Content-Style-Type" content="text/css" />
    <link href="theme/Master.css" rel="stylesheet" type="text/css" />
    <title>Employee list</title>
  </head>
  <body>
    <h1>Employee listing as of
      <%= java.text.DateFormat.getDateInstance().format(new java.util.Date()) %>2
    </h1>
    <sql:dataSourceSpec id="MyConnection"                                   3
            dataSource="jdbc/empdb" />
    <sql:select id="emplist" connectionSpec="MyConnection">                 4
      <sql:sql>
        SELECT EMPNO
             , LASTNAME || ', ' || FIRSTNME || ' ' || MIDINIT
          FROM DSN8710.EMP
         ORDER BY EMPNO
      </sql:sql>
    </sql:select>
    <table>
      <col width="30%" />
      <col width="70%" />
      <thead>
        <tr align="left">
          <th>Serial</th>
          <th>Name</th>
        </tr>
      </thead>
      <tbody>
        <sql:repeat name="emplist" over="rows">                             5
          <tr>
            <td>
              <a href='EmployeeDetail.jsp?empno=<sql:getColumn index="1"/>'> 6
                <sql:getColumn index="1" />
              </a>
```

```
          </td>
          <td><sql:getColumn index="2" /></td>                              7
        </tr>
      </sql:repeat>
    </tbody>
  </table>
</body>
</html>
```

Notes on Example 16-2 on page 284:

1. This directive makes the database access tag library known to the JSP compiler. We use a prefix of sql to access the tags in the library.
2. This is an example of a Java code snippet inside the HTML code. It uses the `java.util.TextFormat` class to format the current date. The resulting string is then included in the content.
3. This is one of the custom tags from the database access tag library. It declares a data source to be used for subsequent operations, which can then refer to that data source using the name given in the ID attribute. The dataSource attribute gives the JNDI name of the data source.
4. Executes an SQL SELECT statement, using the data source declared in step 3.
5. This causes the enclosed content to be generated several times, once for each row.
6. Access the first…
7. … and second column of the current row, embedding them in a `<td>` element.

Of particular interest is the `sql:repeat` tag for iterating over a result set. Essentially, this is a control structure (a `while` loop) masquerading as a JSP tag. As described above, it allows you to iterate over the result set without explicit Java coding.

For a full description and reference of the Database Access custom tag library, refer to the WSAD documentation.

Compared to the Servlet version (see “Creating the EmployeeList Servlet” on page 251), the JSP version is definitely easier to understand and to maintain. Rather than burying HTML code in Java print statements, the JSP version clearly shows the structure of the resulting output document. All the database query and result set iteration is done for you by the custom tag library, so there is no explicit coding at all (apart from the SELECT statement, of course).

16.3 Creating the EmployeeDetail JSP

The EmployeeDetail JSP page should be relatively straightforward now (see Example 16-3). Since the SQL tag library does not have support for LOB data, we cannot re-write the EmployeePicServlet as a JSP, which is why we still use that Servlet from the new JSP.

Example 16-3 EmployeeDetail.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <%@ taglib uri="jspsql" prefix="sql" %>
    <%@ page
    language="java"
    contentType="text/html; charset=ISO-8859-1"
    %>
```

```
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <meta name="GENERATOR" content="IBM WebSphere Studio">
    <meta http-equiv="Content-Style-Type" content="text/css">
    <link href="theme/Master.css" rel="stylesheet" type="text/css">
    <title>EmployeeDetail.jsp</title>
  </head>
  <body>
    <sql:dataSourceSpec id="MyConnection"
      dataSource="jdbc/empdb" />
    <sql:select id="empdetail" connectionSpec="MyConnection" maxRows="1">     1
      <sql:sql>
        SELECT EMPNO
             , LASTNAME || ', ' || FIRSTNME || ' ' || MIDINIT
             , HIREDATE
             , SALARY
          FROM DSN8710.EMP
         WHERE EMPNO = ?
      </sql:sql>
      <sql:parameter position="1" type="CHAR"
        value='<%= request.getParameter("empno") %>' />                     2
    </sql:select>
    <table>
      <tr>
        <td colspan="2">
                                                                             3
          <img src='EmployeePicServlet?empno=<%=request.getParameter("empno") %>' />
        </td>
      </tr>
      <tr>
        <td>Serial:</td>
        <td><sql:getColumn name="empdetail" index="1" /></td>
      </tr>
      <tr>
        <td>Name:</td>
        <td><sql:getColumn name="empdetail" index="2" /></td>
      </tr>
      <tr>
        <td>Hired:</td>
        <td><sql:getColumn name="empdetail" index="3" /></td>
      </tr>
      <tr>
        <td>Salary:</td>
        <td><sql:getColumn name="empdetail" index="4" /></td>
      </tr>
    </table>
  </body>
</html>
```

Notes on Example 16-3 on page 285:

1 We indicate that this is a single-row query using the maxRows attribute.

2 The JSP engine replaces the `<%= ... %>` construct by the value of the enclosed expression, in this case, the value of the parameter ‘empno’ in the HTTP request. The sql:parameter tag supplies a value for the parameter marker in the SQL statement. Note that, rather than using sql:parameter, we could have embedded the expression directly in the statement:

```
WHERE EMPNO = <%= request.getParameter(“empno”) %>
```

However, using a parameter marker is more efficient since it allows DB2 to cache and reuse the prepared statement.

3 To display the employee's image, we still use EmployeePicServlet from "Creating the EmployeePic Servlet" on page 265.

You can test a JSP exactly the same as the Servlets from Chapter 15, "Using Servlets to access DB2" on page 249, by selecting **Run on server**. Remember that, eventually, a JSP is automatically compiled into a Servlet by the Web application server.

16.4 Deploying to WebSphere Application Server

This is similar to deploying a Servlet, but you need to use a different URL from the Web browser.

Part 5

Appendixes

A

SQLSTATE categories

Table 16-1 below provides an overview of selected SQLSTATE categories to help you categorize the cause of an `SQLException`.

For full details, and the individual SQLSTATE values, refer to *DB2 Universal Database for OS/390 and z/OS Messages and Codes*, GC26-9940.

Table 16-1 SQLSTATE categories

Category	Meaning	Example
01	Warning	(reported as SQLWarning on the execution context)
02	No Data	No row found for SELECT INTO
07	Dynamic SQL Error	No value provided for a parameter marker
08	Connection Exception	
09	Triggered Action Exception	
0A	Feature Not Supported	
0F	Invalid Token	
21	Cardinality Violation	More than one row for SELECT INTO
22	Data Exception	String too long for column
23	Constraint Violation	INSERT or UPDATE a NULL value into a NOT NULL column Invalid foreign key value
24	Invalid Cursor State	Cursor is closed
25	Invalid Transaction State	
26	Invalid SQL Statement Identifier	
2D	Invalid Transaction Termination	
34	Invalid Cursor Name	

Category	Meaning	Example
38	External Function Exception	
39	External Function Call Exception	
40	Transaction Rollback	Deadlock or timeout occurred
42	Syntax Error or Access Rule Violation	• Syntax error in SQL statement • No priviledge to perform operation
44	WITH CHECK OPTION Violation	
51	Invalid Application State	Package not found
53	Invalid Operand or Inconsistent Specification	
54	SQL or Product Limit Exceeded	
55	Object Not in Prerequisite State	
56	Miscellaneous SQL or Product Error	
57	Resource Not Available or Operator Intervention	
58	System Error	

B

Source code of sample programs

This appendix provides the source code for the examples presented in this publication.

To avoid having to retype all the code, or even copy/paste it, we also provide the sample source code as additional material that you can download from the Web.
See Appendix C, “Additional material” on page 313, for details.

Hello.java

Here (Example B-1) we provide the listing of the Hello program written with JDBC.

Example: B-1 Hello.java

```
/*
 * Created on Oct 25, 2003
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */

package com.ibm.itso.sg246435.jdbc;
import java.math.BigDecimal;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
/**
* Sample test program to retrieve all employees
* in the EMP sample tables whose salary is in
* a given range.
*
* @author Ulrich Gehlert
*/
public class Hello {
   /** JDBC URL to the database. */
   private static final String url =
      "jdbc:db2://wtsc63.itso.ibm.com:33756/DB7Y"
         + ":retrieveMessagesFromServerOnGetMessage=TRUE;";
   /** User name to connect to the database. */
   private static final String user = "bart";
   /** Password for the database connection. */
   private static final String password = "secret";
   public static void main(String[] args) {
      Connection conn = null;
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         // Load the JDBC driver.
         Class.forName("com.ibm.db2.jcc.DB2Driver");
         // Connect to the database server.
         conn = DriverManager.getConnection(url, user, password);
         // Prepare the SELECT statement.
         stmt =
            conn.prepareStatement(
               "SELECT LASTNAME, FIRSTNME, SALARY"
                  + " FROM DSN8710.EMP"
                  + " WHERE SALARY BETWEEN ? AND ?");
         // Set parameters for the SELECT statement.
         stmt.setBigDecimal(1, new BigDecimal(30000));
         stmt.setBigDecimal(2, new BigDecimal(50000));
         // Execute the query to retrieve a ResultSet.
         rs = stmt.executeQuery();
         // Iterate over the ResultSet.
         while (rs.next()) {
            String lastname = rs.getString(1); // LASTNAME
```

```
            String firstname = rs.getString(2); // FIRSTNME
            BigDecimal salary = rs.getBigDecimal(3); // SALARY

            System.out.println(
               lastname + ", " + firstname + ": $" + salary);
         }
      } catch (SQLException e) {
         // Print exceptions to the console.
         System.err.println(e.getMessage());
      } catch (Exception e) {
         System.err.println(e);
      } finally {
         // Clean up.
         try {
            if (rs != null)
               rs.close();
            if (stmt != null)
               stmt.close();
            if (conn != null)
               conn.close();
         } catch (SQLException ignored) {
         }
      }
   }
}
```

Hello.sqlj

Here (Example B-2) we provide the listing for the first SQLJ program, Hello.sqlj.

Example: B-2 Hello.sqlj

```
package com.ibm.itso.sg246435.sqlj;
//import java.sql.*;
//import sqlj.runtime.ref.*;

import java.math.BigDecimal;
import java.sql.SQLException;
/**
* Sample test program to retrieve all employees
* in the EMP sample tables whose salary is in
* a given range.
*
* @author Ulrich Gehlert
*/
public class Hello {
   #sql public static context Ctx;
   #sql static iterator EmployeeIterator (String, String, BigDecimal);

   /**
   * Load JDBC driver and initialize SQLJ connection context.
   *
   * @return
   * The SQLJ connection context.
   */
   private static Ctx initialize(String url, String user, String password)
```

```
      throws ClassNotFoundException, SQLException {
      Class.forName("com.ibm.db2.jcc.DB2Driver");
      return new Ctx(url, user, password, false);
   }
   public static void main(String[] args) {
      if (args.length != 5) {
         System.err.println(
            "Usage: java "
               + Hello.class.getName()
               + " <url> <user> <password> <min> <max>");
         return;
      }
      BigDecimal min = new BigDecimal(args[3]);
      BigDecimal max = new BigDecimal(args[4]);
      Ctx ctx = null;
      EmployeeIterator iter = null;
      try {
         ctx = initialize(args[0], args[1], args[2]);
         #sql [ctx] iter = {
         SELECT LASTNAME
         , FIRSTNME
         , SALARY
         FROM DSN8710.EMP
         WHERE SALARY BETWEEN :min AND :max
         ORDER BY LASTNAME, FIRSTNME
         };
         String lastname = null;
         String firstname = null;
         BigDecimal salary = null;
         while (true) {
            #sql {
            FETCH :iter
            INTO :lastname
            , :firstname
            , :salary
            };
            if (iter.endFetch())
               break;
            System.out.println(
               lastname + ", " + firstname + ": $" + salary);
         }
      } catch (Throwable e) {
         e.printStackTrace();
      } finally {
         try {
            if (iter != null)
               iter.close();
            if (ctx != null)
               ctx.close();
         } catch (SQLException ignored) {
         }
      }
   }
}
```

Initial SQLJ employee programs

Following are the two programs that we use to do the initial work for the employee class.

Employee.sqlj

Example B-3 is the employee sqlj program. Note that the version that is presented here is what was used in the last chapters of the publication when deploying programs on a Web Application Server. This version varies a lilttle bit from the initial one. However, when following the steps outlined in the publication, it should be easy to determine the differences.

Example: B-3 Employee.sqlj

```
package com.ibm.itso.sg246435.sqlj;

import java.sql.*;
import sqlj.runtime.ref.*;
import java.math.BigDecimal;
import java.sql.Date;
import java.sql.SQLException;
// next one is needed to retrieve pictures
import java.io.InputStream;
// next ones are needed to store pictures
import sqlj.runtime.BinaryStream;
import java.io.IOException;
import java.io.File;
import java.io.FileInputStream;


public class Employee {

   #sql public static context Ctx with (dataSource="jdbc/empdb");

   private final String empNo;
   private String firstName;
   private String middleInitial = " ";
   private String lastName;
   private Date hireDate;
   private Short educationLevel;
   private boolean male;
   private BigDecimal salary;

   /**
    * @return
    */
   public Short getEducationLevel() {
      return educationLevel;
   }

   /**
    * @return
    */
   public String getEmpNo() {
      return empNo;
   }

   /**
    * @return
```

```
 */
public String getFirstName() {
   return firstName;
}

/**
 * @return
 */
public Date getHireDate() {
   return hireDate;
}

/**
 * @return
 */
public String getLastName() {
   return lastName;
}

/**
 * @return
 */
public boolean isMale() {
   return male;
}

/**
 * @return
 */
public String getMiddleInitial() {
   return middleInitial;
}

/**
 * @return
 */
public BigDecimal getSalary() {
   return salary;
}

/**
 * @param short1
 */
public void setEducationLevel(Short short1) {
   educationLevel = short1;
}

/**
 * @param string
 */
public void setFirstName(String string) {
   firstName = string;
}

/**
 * @param date
 */
public void setHireDate(Date date) {
   hireDate = date;
}
```

```
/**
 * @param string
 */
public void setLastName(String string) {
   lastName = string;
}

/**
 * @param b
 */
public void setMale(boolean b) {
   male = b;
}

/**
 * @param string
 */
public void setMiddleInitial(String string) {
   middleInitial = string;
}

/**
 * @param decimal
 */
public void setSalary(BigDecimal decimal) {
   salary = decimal;
}
/**
    * @param String
    */
public Employee(String empNo) {
   this.empNo = empNo;
}
/**
* toString - return printable representation
*/
public String toString() {
   return getLastName() + ", " + getFirstName() + " " + getMiddleInitial();
}

public void insert(Ctx ctx) throws SQLException {
   #sql [ctx] {
   INSERT INTO DSN8710.EMP (
   EMPNO
   , FIRSTNME
   , MIDINIT
   , LASTNAME
   , HIREDATE
   , SEX
   , SALARY
   ) VALUES (
   :empNo
   , :firstName
   , :middleInitial
   , :lastName
   , :hireDate
   , :(male ? "M" : "F")
   , :salary
   )
```

```
    };
}

public static Employee findByPrimaryKey(Ctx ctx, String empNo)
    throws SQLException {
    Employee emp = new Employee(empNo);
    #sql [ctx] {
    SELECT FIRSTNME
    , MIDINIT
    , LASTNAME
    , HIREDATE
    , CASE SEX WHEN 'M' THEN 1 ELSE 0 END
    , SALARY
    INTO :(emp.firstName)
    , :(emp.middleInitial)
    , :(emp.lastName)
    , :(emp.hireDate)
    , :(emp.male)
    , :(emp.salary)
    FROM DSN8710.EMP
    WHERE EMPNO = :empNo
    };
    return emp;
}

public void delete(Ctx ctx) throws SQLException {
    #sql [ctx] {
    DELETE FROM DSN8710.EMP
    WHERE EMPNO = :empNo
    };
}

public void update(Ctx ctx) throws SQLException {
    #sql [ctx] {
    UPDATE DSN8710.EMP
    SET FIRSTNME = :firstName
    , LASTNAME = :lastName
    , MIDINIT = :middleInitial
    , HIREDATE = :hireDate
    , SEX = :(male ? "M" : "F")
    , SALARY = :salary
    WHERE EMPNO = :empNo
    };
}

#sql public static iterator EmployeeIterator (
  String // EMPNO
, String // FIRSTNME
, String // MIDINIT
, String // LASTNAME
, Date // HIREDATE
, boolean // SEX
, BigDecimal // SALARY
);

public static EmployeeIterator findAll(Ctx ctx) throws SQLException {
    EmployeeIterator iter;
    #sql [ctx] iter = {
    SELECT
      EMPNO
```

```
   , FIRSTNME
   , MIDINIT
   , LASTNAME
   , HIREDATE
   , CASE SEX WHEN 'M' THEN 1 ELSE 0 END
   , SALARY
   FROM DSN8710.EMP
   };
   return iter;
}

public static Employee fetch(EmployeeIterator iter) throws SQLException {
   String empno = null;
   String firstName = null;
   String middleInitial = null;
   String lastName = null;
   Date hireDate = null;
   boolean male = false;
   BigDecimal salary = null;
   #sql {
   FETCH :iter
   INTO :empno
   , :firstName
   , :middleInitial
   , :lastName
   , :hireDate
   , :male
   , :salary
   };
   if (iter.endFetch())
      return null;
   Employee emp = new Employee(empno);
   emp.firstName = firstName;
   emp.lastName = lastName;
   emp.middleInitial = middleInitial;
   emp.hireDate = hireDate;
   emp.male = male;
   emp.salary = salary;
   return emp;
}

/**
* Retrieves the employee's picture.
*
* @return
* An InputStream to read the picture in GIF format,
* or <code>null</code> if no picture is available.
*
* @exception SQLException
* A database error occurred.
*/
public InputStream getPicture(Ctx ctx) throws SQLException {
   Blob picture;
   try {
      #sql [ctx] {
      SELECT BMP_PHOTO
      INTO :picture
      FROM DSN8710.EMP_PHOTO_RESUME
      WHERE EMPNO = :empNo
      };
```

```
      } catch (SQLException sqlException) {
         if (sqlException.getSQLState().equals("02000")) // row not found
            return null;
         else
            throw sqlException;
      }
      return picture.getBinaryStream();
   }


   /**
   * Creates the employee's picture.
   *
   * @param picture
   * An InputStream supplying the picture in BMP format.
   * On return from this method, the stream is closed.
   *
   * @exception SQLException
   * A database error occurred.
   * @exception IOException
   * An error occurred reading the input stream.
   */
   public void createPicture(Ctx ctx, InputStream picture, int length)
   throws SQLException, IOException
   {
   try {
   BinaryStream lobStream = new BinaryStream(picture, length);
   #sql [ctx] {
   INSERT INTO DSN8710.EMP_PHOTO_RESUME (
   EMPNO
   , BMP_PHOTO
   ) VALUES (
   :empNo
   , :lobStream
   )
   };
      }
   catch (SQLException e) {
            System.err.println(e + ". SQLCODE = " + e.getErrorCode());
         } catch (Exception e) {
            System.err.println(e);
         }
      finally {
   picture.close();
   }
   }
   /**
   * Creates the employee's picture.
   *
   * @param pictureFilename
   * Name of a file containing the picture in BMP format.
   *
   * @exception SQLException
   * A database error occurred.
   * @exception IOException
   * An error occurred reading the file.
   */
   public void createPicture(Ctx ctx, String pictureFilename) throws SQLException,
IOException
   {
```

```
   File pictureFile = new File(pictureFilename);
   createPicture(ctx, new FileInputStream(pictureFile), (int) pictureFile.length());
   }



}
```

EmployeeTest.java

Example B-4 is the program that is used for testing the Employee class listed before. We recommend commenting and uncommenting different sections of the program when running it, to have better control over its execution.

Example: B-4 EmployeeTest.java

```
package com.ibm.itso.sg246435.sqlj;
import java.io.File; // not in here before
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.math.BigDecimal;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;
import com.ibm.itso.sg246435.sqlj.Employee.EmployeeIterator;

public class EmployeeTest {
   private static Employee.Ctx ctx;
   /**
   * Load JDBC driver and initialize SQLJ connection context.
   *
   * Driver name, URL and driver-specific properties
   * are read from a configuration file.
   */
   private static void initialize()
      throws
         ClassNotFoundException,
         SQLException,
         FileNotFoundException,
         IOException {
      // Read the properties file
      FileInputStream propsFile =
         new FileInputStream("EmployeeTest.properties");
      Properties properties = new Properties();
      properties.load(propsFile);
      propsFile.close();
      // Get driver property, and load the driver
      String driver =
         properties.getProperty("driver", "com.ibm.db2.jcc.DB2Driver");
      Class.forName(driver);
      String url = properties.getProperty("url");
      Connection conn = DriverManager.getConnection(url, properties);
```

```
   // Set up the connection context with autocommit disabled
   ctx = new Employee.Ctx(conn);
}
public static void testInsert() throws SQLException {
   Employee emp = new Employee("000042");
   emp.setFirstName("Arthur");
   emp.setLastName("Dent");
   emp.setMale(true);
   emp.setSalary(new BigDecimal(200000)); // Wish it were so...
   emp.insert(ctx);
   System.out.println("Employee " + emp + " created successfully.");
}

public static void testFindAll() throws SQLException {
EmployeeIterator iter = null;
try {
iter = Employee.findAll(ctx);
for (;;) {
Employee emp = Employee.fetch(iter);
if (emp == null)
{  //  iter.close();
    break; }
System.out.println(emp);
}
} finally {
if (iter != null) iter.close();
}
}


public static void testGetPicture(String empno, File dir) throws SQLException,
IOException {
Employee emp = Employee.findByPrimaryKey(ctx,empno);
InputStream in = null;
OutputStream out = null;
try {
in = emp.getPicture(ctx);
if (in == null) {
System.out.println("No picture available for employee " + empno);
} else {
out = new FileOutputStream(new File(dir, "Emp" + empno + ".bmp"));
int nread;
byte[] buf = new byte[1024];
while ((nread = in.read(buf)) > 0)
out.write(buf, 0, nread);
System.out.println("Picture retrieved for employee " + empno + " in "
                    + dir + " with name Emp" + empno + ".bmp");
}
} finally {
if (in != null) in.close();
if (out != null) out.close();
}
}



public static void main(String[] args) {
   try {
      initialize();
      testInsert();
```

```
            Employee emp = Employee.findByPrimaryKey(ctx, "000042");
            System.out.println("Successfully retrieved " + emp);
            emp.setLastName("Dentist");
            emp.update(ctx);
            System.out.println("Successfully updated " + emp);
//          emp.createPicture(ctx,"C:/temp/arthur_dent.bmp");
//          System.out.println("Successfully stored picture of " + emp);
//          testGetPicture("000042",new File("c:/temp"));
            emp.delete(ctx);
            try {
            emp = Employee.findByPrimaryKey(ctx, "000042");
            } catch (SQLException expected) {
            if (expected.getSQLState().equals("02000")) {
            System.out.println("Employee record deleted successfully");
            } else {
             //  Different SQL exception -- we didn't really expect it
            throw expected;
            }
            }
            testFindAll();
            testGetPicture("000130",new File("c:/temp"));
         } catch (SQLException e) {
            System.err.println(e + ". SQLCODE = " + e.getErrorCode());
         } catch (Exception e) {
            System.err.println(e);
         }
      }
}
```

Servlets

We also provide listings for the three Servlets that are used in this publication. Again, the version presented here is the final version, that is, after all Servlets are linked to each other.

EmployeeListServlet

Example B-5 shows the EmployeeListServlet.

Example: B-5 EmployeeListServlet

```
package com.ibm.itso.sg246435.web;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.SQLException;

import javax.servlet.Servlet;
import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ibm.itso.sg246435.sqlj.Employee;

/**
```

```
 * @version 1.0
 * @author
 */
public class EmployeeListServlet extends HttpServlet implements Servlet {

   /**
   * @see javax.servlet.http.HttpServlet#void (javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
   */

   public void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {
      PrintWriter out = resp.getWriter();
      out.println(
         "<!DOCType HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>");
      out.println("<HTML>");
      out.println("<HEAD>");
      out.println("<TITLE>Employee listing</TITLE>");
      out.println("</HEAD>");
      out.println("<BODY>");
      out.println("<H1>Employee listing</H1>");
      try {
         printTable(out);
      } catch (SQLException e) {
         throw new ServletException(e.getMessage(), e);
      }
      out.println("</BODY>");
      out.println("</HTML>");
      out.close();
   }

   /**
   * Print the supplied value to the servlet output stream
   * embedded in a <code>td</code> element.
   *
   * @param out
   * The servlet output stream.
   * @param value
   * The value to be printed.
   */
   private void printCol(PrintWriter out, String value) {
      out.println("<td>");
      out.println(value);
      out.println("</td>");
   }

   /**
   * Print employee data to the servlet output stream
   * in an HTML table row.
   *
   * @param out
   * The servlet output stream.
   * @param emp
   * The employee to be displayed.
   */
   private void printRow(PrintWriter out, Employee emp) {
      out.println("<tr>");
      printCol(out, "<a href='EmployeeDetailServlet?empno=" + emp.getEmpNo() + "'>" +
      emp.getEmpNo() + "</a>");
//    printCol(out, emp.getEmpNo());
```

```
      printCol(out, emp.getLastName());
      printCol(out, emp.getFirstName() + ' ' + emp.getMiddleInitial());
      out.println("</tr>");
   }
   private static Employee.Ctx ctx;

   /**
   * Prints the table of employee data.
   * *
   * @param out
   * The servlet output stream.
   *
   * @exception SQLException
   * A database error occurred.
   */
   private void printTable(PrintWriter out) throws SQLException {
      out.println("<table>");
      ctx = new Employee.Ctx();
      Employee.EmployeeIterator iter = Employee.findAll(ctx);
      try {
         Employee emp;
         while ((emp = Employee.fetch(iter)) != null)
            printRow(out, emp);
         out.println("</table>");
      } finally {
         iter.close();
      }
   }

}
```

EmployeeDetailServlet

Example B-6 shows the EmployeeDetailServlet code.

Example: B-6 EmployeeDetailServlet

```
package com.ibm.itso.sg246435.web;

import java.io.IOException;
import java.io.PrintWriter; // added

import javax.servlet.Servlet;
import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ibm.itso.sg246435.sqlj.Employee; //added

/**
 * @version 1.0
 * @author
 */
public class EmployeeDetailServlet extends HttpServlet implements Servlet {

   private static Employee.Ctx ctx;
```

```
   /**
   * @see javax.servlet.http.HttpServlet#void (javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
   */
   public void doGet(HttpServletRequest req, HttpServletResponse resp)
   throws ServletException, IOException {
   PrintWriter out = resp.getWriter();
// String empno = "000130";
   String empno = req.getParameter("empno");
   out.println("<!DOCType HTML PUBLIC '-//W3C//DTD HTML 4.01 Transitional//EN'>");
   out.println("<html>");
   out.println("<head><title>Employee detail</title></head>");
   out.println("<body>");
   out.println("<table>");
   try {
   ctx = new Employee.Ctx();
   Employee emp = Employee.findByPrimaryKey(ctx,empno);
   out.println("<tr><img src='EmployeePicServlet?empno=" + empno + "'></tr>");
   printRow(out, "Serial", emp.getEmpNo());
   printRow(out, "Name", emp.getLastName() + ", " + emp.getFirstName());
   printRow(out, "Hired", emp.getHireDate());
   printRow(out, "Salary", emp.getSalary());
   } catch (Exception e) {
   throw new ServletException(e);
   }
   out.println("</table>");
   out.println("</body>");
   out.println("</html>");
   }
   /**
   * Prints one row of employee information to the
   * servlet output stream. The left column shows a
   * label, the right column the actual data.
   *
   * @param out
   * The servlet response output stream.
   * @param label
   * The label to be displayed in the left column.
   * @param value
   * The value to be displayed in the right column.
   */
   private void printRow(PrintWriter out, String label, Object value) throws IOException {
   out.println("<tr>");
   out.println("<td>" + label + ":</td>");
   out.println("<td>" + value + "</td>");
   out.println("</tr>");
   }

}
```

EmployeePicServlet

Example B-7 on page 309 shows the code for the EmployeePicServlet that is used to retrieve an employee's picture.

Example: B-7 EmployeePicServlet

```
package com.ibm.itso.sg246435.web;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import javax.servlet.Servlet;
import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ibm.itso.sg246435.sqlj.Employee;

/**
 * @version 1.0
 * @author
 */
public class EmployeePicServlet extends HttpServlet implements Servlet {

   /**
   * @see javax.servlet.http.HttpServlet#void (javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
   */
   private static Employee.Ctx ctx;

   public void doGet(HttpServletRequest req, HttpServletResponse resp)
   throws ServletException, IOException {
   resp.setContentType("image/bmp");
   OutputStream out = null;
   InputStream pic = null;
// String empno = "000130";
   String empno = req.getParameter("empno");
   try {
   ctx = new Employee.Ctx();
   Employee emp = Employee.findByPrimaryKey(ctx,empno);
   out = resp.getOutputStream();
   pic = emp.getPicture(ctx);
// copy(pic, out);
   if (pic == null) {
   getServletContext().getRequestDispatcher("/images/noimage.gif").forward(req, resp);
   } else {
   copy(pic, out);
   }
   } catch (Exception e) {
   throw new ServletException(e);
   } finally {
//    Clean up
   if (pic != null) pic.close();
   if (out != null) out.close();
   }
   }
   private void copy(InputStream in, OutputStream out) throws IOException {
   byte[] buf = new byte[1024];
   int nread;
   while ((nread = in.read(buf)) > 0)
   out.write(buf, 0, nread);
```

```
    }
}
```

JavaServer Pages

Here we list both JSPs that we developed in this publication.

EmployeeList JPS

Example B-8 shows the source code for the EmployeeList JSP program.

Example: B-8 EmployeeList

```
<!DOCType HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<%@ taglib uri="jspsql" prefix="sql" %>
<%@ page
language="java"
contentType="text/html; charset=ISO-8859-1"
%>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<meta name="GENERATOR" content="IBM WebSphere Studio" />
<meta http-equiv="Content-Style-Type" content="text/css" />
<link href="theme/Master.css" rel="stylesheet" type="text/css" />
<title>Employee list</title>
</head>
<body>
<h1>Employee listing as of <%=java.text.DateFormat.getDateInstance().format(new
java.util.Date())%>
</h1>
<sql:dataSourceSpec id="MyConnection"
   dataSource="jdbc/empdb" />
<sql:select id="emplist" connectionSpec="MyConnection">
   <sql:sql>
SELECT EMPNO
, LASTNAME || ', ' || FIRSTNME || ' ' || MIDINIT
FROM DSN8710.EMP
ORDER BY EMPNO
</sql:sql>
</sql:select>
<table>
   <col width="30%" />
   <col width="70%" />
   <thead>
      <tr align="left">
         <th>Serial</th>
         <th>Name</th>
      </tr>
   </thead>
   <tbody>
      <sql:repeat name="emplist" over="rows">
         <tr>
            <td><a href='EmployeeDetail.jsp?empno=<sql:getColumn index="1"/>'>
<sql:getColumn
```

```
                index="1" /> </a></td>
             <td><sql:getColumn index="2" /></td>
          </tr>
       </sql:repeat>
    </tbody>
</table>
</body>
</html>
```

EmployeeDetail.JSP

Example B-9 shows the source code of the EmployeeDetail JSP. Note that this one invokes the EmployeePicServlet as part of its processing.

Example: B-9 EmployeeDetail

```
<!DOCType HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<%@ taglib uri="jspsql" prefix="sql" %>
<%@ page
language="java"
contentType="text/html; charset=ISO-8859-1"
%>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<meta name="GENERATOR" content="IBM WebSphere Studio">
<meta http-equiv="Content-Style-Type" content="text/css">
<link href="theme/Master.css" rel="stylesheet" type="text/css">
<title>EmployeeDetail.jsp</title>
</head>
<body>
<sql:dataSourceSpec id="MyConnection"
dataSource="jdbc/empdb" />
<sql:select id="empdetail" connectionSpec="MyConnection" maxRows="1">
<sql:sql>
SELECT EMPNO
, LASTNAME || ', ' || FIRSTNME || ' ' || MIDINIT
, HIREDATE
, SALARY
FROM DSN8710.EMP
WHERE EMPNO = ?
</sql:sql>
<sql:parameter position="1" type="CHAR"
value='<%= request.getParameter("empno") %>' />
</sql:select>
<table>
<tr>
<td colspan="2">
<img src='EmployeePicServlet?empno=<%=request.getParameter("empno") %>' />
</td>
</tr>
<tr>
<td>Serial:</td>
<td><sql:getColumn name="empdetail" index="1" /></td>
</tr>
<tr>
<td>Name:</td>
```

```
<td><sql:getColumn name="empdetail" index="2" /></td>
</tr>
<tr>
<td>Hired:</td>
<td><sql:getColumn name="empdetail" index="3" /></td>
</tr>
<tr>
<td>Salary:</td>
<td><sql:getColumn name="empdetail" index="4" /></td>
</tr>
</table>
</body>
</html>
```

C

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

```
ftp://www.redbooks.ibm.com/redbooks/SG246435
```

Alternatively, you can go to the IBM Redbooks Web site at:

```
ibm.com/redbooks
```

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246435.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

File name	*Description*
SG246435.zip	Zipped code samples

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

When unzipping the file, a directory structure matching the java package names is created, as well as the naming structure used in WSAD to store the projects.

The folder should have the followning content:

noimage.gif File that gets loaded when no picture exists in the database

Hello.java	Our first JDBC program
Hello.sqlj	Our first SQLJ program
Employee.sqlj	Our Employee class
EmployeeTest.java	Employee class test program
EmployeeListServlet.java	Servlet listing all employees
EmployeeDetailServlet.java	Servlet lising some details of a certain employee
EmployeePicServlet.java	Servlet retrieving an employee's picture
EmployeeList.jsp	JSP listing all employees
EmployeeDetail.jsp	JSP providing details about an individual employee
Spufi.java	Poor man's SPUFI
Getopt.java	A class for parsing command line arguments passed to programs

Abbreviations and acronyms

AIX	Advanced Interactive eXecutive from IBM
APAR	Authorized Program Analysis Report
APAR	Authorized program analysis report
AR	Application Requester
ARM	Automatic restart manager
AS	Application Server
ASCII	American National Standard Code for Information Interchange
BLOB	Binary large objects
CA	Coordinator Agent
CCA	Client configuration assistant
CCSID	Coded character set identifier
CD	Compact disk
CDRA	Character Data Representation Architecture
CEC	Central electronics complex
CF	Coupling facility
CFCC	Coupling facility control code
CFRM	Coupling facility resource management
CGI	Common Gateway Interface
CICS	Customer Information control System
CLI	Call level interface
CLP	Command line processor
CPU	Central Processing Unit
CSA	Common storage area
CTT	Created temporary table
DASD	Direct access storage device
DB2 Connect EE	DB2 Connect Enterprise Edition
DB2 Connect PE	DB2 Connect Personal Edition
DB2 PM	DB2 performance monitor
DB2 UDB	DB2 Universal Database
DB2RA	DB2 Request Access
DBAT	Database access thread
DBD	Database descriptor
DBID	Database identifier
DBMS	Database management system
DBRM	Database request module
DCL	Data control language
DDCS	Distributed database connection services
DDF	Distributed Data Facility
DDL	Data Definition Language
DDM	Distributed Data Management
DLL	Dynamic load library manipulation language
DML	Data manipulation language
DNS	Domain name server
DRDA	Distributed Relational Database Architecture
DSC	Dynamic statement cache, local or global
DTT	Declared temporary tables
DUW	Distributed Unit of Work
EA	Extended addressability
EBCDIC	Extended binary coded decimal interchange code
ECS	Enhanced catalog sharing
ECSA	Extended common storage area
EDM	Environment descriptor management
ERM	Enterprise resource management
ESA	Enterprise Systems Architecture
ESP	Enterprise Solution Package
ETR	External throughput rate, an elapsed time measure, focuses on system capacity
FD:OCA	Formatted Data Object Content
FTD	Functional track directory
FTP	File Transfer Program
GB	Gigabyte (1,073,741,824 bytes)
GBP	Group buffer pool
GRS	Global resource serialization
GUI	Graphical user interface
HPJ	High performance Java
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
I/O	Input/output
IBM	International Business Machines Corporation
ICF	Integrated catalog facility

ICMF	Internal coupling migration facility
IFI	Instrumentation Facility Interface
IPLA	IBM Program Licence Agreement
IRLM	Internal resource lock manager
ISPF	Interactive system productivity facility
ISV	Independent software vendor
IT	Information Technology
ITR	Internal throughput rate, a processor time measure, focuses on processor capacity
ITSO	International Technical Support Organization
IVP	Installation verification process
J2EE	Java 2 Platform, Enterprise Edition
J2ME	Java 2 Platform, Micro Edition
J2SE	Java 2 Platform, Standard Edition
JAAS	Java Authentication and Authorization Service
JCE	Java Cryptography Extension
JCC	DB2 Universal Driver for Java Common Connectivity
JDBC	Java DataBase Connectivity
JGSS	Java Generic Security Service
JNDI	Java Naming and Directory Interface
JSP	JavaServer Page
JVM	Java Virtual Machine
KB	Kilobyte (1,024 bytes)
LA	Logical Agent
LE	Language Environment®
LOB	Large object
LPAR	Logical partition
LPL	Logical page list
LRECL	Logical record length
LRSN	Log record sequence number
LUW	Logical unit of work
MB	Megabyte (1,048,576 bytes)
NPI	Non-partitioning index
ODBC	Open Data Base Connectivity
OS/390	Operating System/390®
PAV	Parallel access volume
PDS	Partitioned data set
PIB	Parallel index build
PSID	Pageset identifier
PSP	Preventive service planning
PTF	Program temporary fix
PUNC	Possibly uncommitted
QA	Quality Assurance
QMF	Query Management Facility
RACF	Resource Access Control Facility
RBA	Relative byte address
RDBMS	Relational Database Management System
RECFM	Record format
RID	Record identifier
RR	Repeatable read
RRS	Resource recovery services
RRSAF	Resource recovery services attach facility
RS	Read stability
RUW	Remote Unit of Work
SAA®	System Application Architecture
SDK	Software development kit
SPUFI	SQL Processing Using File Input
SQL	Structured Query Language
SRB	Service Request Block
SU	Service Unit
TCP/IP	Transmission Control Protocol/Internet Protocol
UOW	Unit Of Work
URL	Uniform Resource Locator
USS	UNIX System Service
WAS	WebSphere Application Server
WSAD	WebSphere Studio Application Developer
XML	Extensible Markup Language

Glossary

Access path The method that is selected by the optimizer for retrieving data from a specific table. For example, an access path can involve the use of an index, a sequential scan, or a combination of the two.

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

ANSI. American National Standards Institute.

API. See *Application Program Interface.*

Applet See Java Applet**.**

Application (1) A program or set of programs that perform a task; for example, a payroll application. (2) In Java programming, a self-contained, stand-alone Java program that includes a static main method. It does not require an applet viewer. Contrast with applet.

Application plan The control structure produced during the bind process and used by DB2 to process SQL statements encountered during statement execution.

Application program interface (API) A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or licensed program.

Application requester (AR) See requester.

AR Application requester. See requester.

ASCII (1) American Standard Code for Information Interchange. A standard assignment of 7-bit numeric codes to characters. See also *Unicode*. (2) An encoding scheme used to represent strings in many environments, typically on PCs and workstations. Contrast with EBCDIC.

Attachment facility An interface between DB2 UDB for OS/390 and TSO, IMS, CICS, or batch address spaces. An attachment facility allows application programs to access DB2 UDB for OS/390.

Authorization ID A string that can be verified for connection to DB2 and to which a set of privileges are allowed. It can represent an individual, an organizational group, or a function, but DB2 does not determine this representation.

Autocommit To automatically commit the current unit of work after each SQL statement.

Automatic bind. (More correctly automatic rebind). A process by which SQL statements are bound automatically (without a user issuing a BIND command) when an application process begins execution and the bound application plan or package it requires is not valid.

Base table (1) A table created by the SQL CREATE TABLE statement that is used to hold persistent data. Contrast with result table and temporary table. (2) A table containing a LOB column definition. The actual LOB column data is not stored along with the base table. The base table contains a row identifier for each row and an indicator column for each of its LOB columns. Contrast with auxiliary table.

Bean A definition or instance of a JavaBeans component. See *JavaBeans*.

Binary large object (BLOB) A sequence of bytes with a size ranging from 0 bytes to 2 gigabytes. This string does not have an associated code page and character set. Image, audio, and video objects are stored in BLOBs. Compare to *character large object* (CLOB).

Bind The process by which the output from the DB2 precompiler is converted to a usable control structure called a package or an application plan. During the process, access paths to the data are selected and some authorization checking is performed.

BLOB See *Binary large object*.

Browser An Internet-based tool that lets users browse Web sites.

Built-in function A function that is supplied by DB2. Contrast with user-defined function.

Bytecode Machine-independent code generated by the Java compiler and executed by the Java interpreter.

CAF See *call attachment facility.*

Call attachment facility (CAF) A DB2 attachment facility for application programs running in TSO or MVS batch. The CAF is an alternative to the DSN command processor and allows greater control over the execution environment.

Call level interface (CLI) A callable application program interface (API) for database access, which is an alternative to using embedded SQL. In contrast to embedded SQL, DB2 CLI does not require the user to precompile or bind applications, but instead provides a standard set of functions to process SQL statements and related services at run time.

Casting Explicitly converting an object's or primitive's data type.

Catalog In DB2, a collection of tables that contains descriptions of objects such as tables, views, and indexes.

Catalog table Any table in the DB2 catalog.

CGI The Common Gateway Interface (CGI) is a means of allowing a Web server to execute a program that you provide rather than to retrieve a file. A number of popular Web servers support the CGI. For some applications, for example, displaying information from a database, you must do more than simply retrieve an HTML document from a disk and send it to the Web browser. For such applications, the Web server has to call a program to generate the HTML to be displayed. The CGI is not the only such interface, however.

Character large object (CLOB) A sequence of characters (single-byte, multibyte, or both) up to 2 gigabytes. A CLOB can be used to store large text objects. Also called character large object string. Compare to *Binary large object* (BLOB).

Class An encapsulated collection of data and methods to operate on the data. A class may be instantiated to produce an object that is an instance of the class.

Class hierarchy The relationships between classes that share a single inheritance. All Java classes inherit from the Object class.

Class method Methods that apply to the class as a whole rather than its instances (also called a *static method*).

Class path When running a Java , a list of directories and JAR files that contain resource files or Java classes that a program can load dynamically at run time.

Class variable Variables that apply to the class as a whole rather than its instances (also called a *static field*).

CLASSPATH In your deployment environment, the environment variable keyword that specifies the directories and Jar files in which to look for class and resource files.

CLI See call level interface.

CLOB See *Character large object*.

Codebase An attribute of the <APPLET> tag that provides the relative path name for the classes. Use this attribute when your class files reside in a different directory than your HTML files.

Column function An SQL operation that derives its result from a collection of values across one or more rows. Contrast with scalar function.

Commit The operation that ends a unit of work by releasing locks so that the database changes made by that unit of work can be perceived by other processes.

Common Connector Framework In the Enterprise Access Builder, interface and class definitions that provide a consistent means of interacting with enterprise resources (for example, CICS and Encina® transactions) from any Java execution environment.

Connection handle The data object that contains information associated with a connection managed by DB2 CLI. This includes general status information, transaction status, and diagnostic information.

Cursor A named control structure used by an application program to point to a row of interest within some set of rows, and to retrieve rows from the set, possibly making updates or deletions.

Cursor stability (CS) An isolation level that locks any row accessed by a transaction of an application while the cursor is positioned on the row. The lock remains in effect until the next row is fetched or the transaction is terminated. If any data is changed in a row, the lock is held until the change is committed to the database.

Data Definition Language (DDL) A language for describing data and its relationships in a database.

Data Manipulation Language (DML) A subset of SQL statements used to manipulate data.

Database management system (DBMS) A software system that controls the creation, organization, and modification of a database and access to the data stored within it.

Database request module (DBRM) A data set member that is created by the DB2 UDB for OS/390 precompiler and that contains information about SQL statements. DBRMs are used in the bind process.

DB2 thread The DB2 structure that describes an application's connection, traces its progress, processes resource functions, and delimits its accessibility to DB2 resources. and services.

DBCLOB A sequence of bytes representing double-byte characters where the size can be up to 2 gigabytes. Although the size of double-byte character large object values can be anywhere up to 2 gigabytes, in general, they are used whenever a double-byte character string might exceed the limits of the VARGRAPHIC type.

DBMS Database management system.

DBRM See *Database Request Module*.

DDF See *Distributed Data Facility*.

DDL See *Data Definition Language*.

Distributed Data Facility (DDF) A set of DB2 UDB for OS/390 components through which DB2 UDB for OS/390 communicates with another RDBMs.

Distributed relational database architecture (DRDA) A connection protocol for distributed relational database processing that is used by IBM's relational database products. DRDA includes protocols for communication between an application and a remote relational database management system, and for communication between relational database management systems.

DLL (dynamic link library) A file containing executable code and data bound to a program at load time or run time, rather than during linking. The code and data in a dynamic link library can be shared by several applications simultaneously. The DLLs. Enterprise Access Builders also generate platform-specific DLLs for the workstation and OS/390 platforms.

DML See *Data Manipulation Language*.

Double precision A floating-point number that contains 64 bits. See also *single precision*.

Double-byte character large object (DBCLOB) See DBCLOB.

DRDA Distributed relational database architecture.

Dynamic bind. A process by which SQL statements are bound as they are entered.

Dynamic SQL SQL statements that are prepared and executed within an application program while the program is executing. In dynamic SQL, the SQL source is contained in host language variables rather than being coded into the application program. The SQL statement can change several times during the application program's execution.

Dynamic Web content Programming elements, such as JavaServer Pages, Servlets, and scripts that require client or server-side processing for accurate run-time rendering in a Web browser.

EBCDIC Extended binary coded decimal interchange code. An encoding scheme used to represent character data in the MVS, VM, VSE, and OS/400Ñ environments. Contrast with ASCII.

Embedded SQL SQL statements coded within an application program. See static SQL.

Enclave In Language Environment for MVS & VM, an independent collection of routines, one of which is designated as the main routine. An enclave is similar to a program or run unit.

Enterprise Java Includes Enterprise JavaBeans as well as open API specifications for: database connectivity, naming and directory services, CORBA/IIOP interoperability, pure Java distributed computing, messaging services, managing system and network resources, and transaction services.

Enterprise JavaBeans A cross-platform component architecture for the development and deployment of multi-tier, distributed, scalable, object-oriented Java applications.

Exception An exception is an object that has caused some sort of new condition, such as an error. In Java, *throwing* an exception means passing that object to an interested party; a signal indicates what kind of condition has taken place. *Catching* an exception means receiving the sent object. *Handling* this exception usually means taking care of the problem after receiving the object, although it might mean doing nothing (which would be bad programming practice).

Extends A subclass or interface extends a class or interface if it add fields or methods, or overrides its methods. See also *derived type.*

External function A function for which the body is written in a programming language that takes scalar argument values and produces a scalar result for each invocation. Contrast with sourced function and built-in function.

Extranet In some cases intranets have connections to other independent intranets. An example would be one company connecting its intranet to the intranet of one of its suppliers. Such a connection of intranets is called an extranet. Depending on the implementation, they may or may not be fully or partially visible to the outside.

Field A data object in a class; for example, a variable.

File Transfer Protocol (FTP) In the Internet suite of protocols, an application layer protocol that uses TCP and Telnet services to transfer bulk-data files between machines or hosts.

First tier The client; the hardware and software with which the end user interacts.

Foreign key A key that is specified in the definition of a referential constraint. Because of the foreign key, the table is a dependent table. The key must have the same number of columns, with the same descriptions, as the primary key of the parent table.

FTP See *File Transfer Protocol*.

Function A specific purpose of an entity or its characteristic action such as a column function or scalar function. (See column function and scalar function.). Furthermore, functions can be user-defined, built-in, or generated by DB2. (See built-in function, cast function, user-defined function, external function, sourced function.)

Garbage collection Java's ability to clean up inaccessible unused memory areas ("garbage") on the fly. Garbage collection slows performance, but keeps the machine from running out of memory.

Hierarchy The order of inheritance in object-oriented languages. Each class in the hierarchy inherits attributes and behavior from its superclass, except for the top-level Object class.

HTTPS HTTPS is a de facto standard developed by Netscape for making HTTP flows secure. Technically, it is the use of HTTP over SSL.

Hypertext Markup Language (HTML) A file format, based on SGML, for hypertext documents on the Internet. Allows for the embedding of images, sounds, video streams, form fields and simple text formatting. References to other objects are embedded using URLs, enabling readers to jump directly to the referenced document.

Hypertext Transfer Protocol (HTTP) The Internet protocol, based on TCP/IP, used to fetch hypertext objects from remote hosts.

IDE See *Integrated Development Environment*.

Identity Column A column that provides a way for DB2 to automatically generate a numeric value for each row that is inserted into the table.

IIOP See Internet Inter-ORB Protocol.

incremental bind. A process by which SQL statements are bound during the execution of an application process, because they could not be bound during the bind process, and VALIDATE(RUN) was specified.

Integrated Development Environment (IDE) A set of software development tools such as source editors, compilers, and debuggers, that are accessible from a single user interface. In WebSphere Studio, the IDE is called the workbench.

Internet The vast collection of interconnected networks that use TCP/IP and evolved from the ARPANET of the late 1960s and early 1970s. The number of independent networks connected into this vast global net is growing daily.

Internet Inter-ORB Protocol (IIOP) A protocol used for communication between Common Object Request Broker Architecture (CORBA) object request brokers.

Internet Protocol (IP) In the Internet suite of protocols, a connectionless protocol that routes data through a network or interconnected networks. IP acts as an intermediary between the higher protocol layers and the physical network. However, this protocol does not provide error recovery and flow control and does not guarantee the reliability of the physical network.

interpreter A tool that translates and executes code line-by-line.

Intranet A private network inside a company or organization that uses the same kinds of software that you would find on the Internet, but that are only for internal use. As the Internet has become more popular, many of the tools used on the Internet are being used in private networks, for example, many companies have Web servers that are available only to employees.

IP See *Internet Protocol*.

JAR file format JAR (Java Archive) is a platform-independent file format that aggregates many files into one. Multiple Java applets and their requisite components (.class files, images, sounds and other resource files) can be bundled in a JAR file and subsequently downloaded to a browser in a single HTTP transaction.

Java An object-oriented programming language for portable, interpretive code that supports interaction among remote objects. Java was developed and specified by Sun Microsystems, Incorporated. The Java environment consists of the JavaOS, the Virtual Machines for various platforms, the object-oriented Java programming language, and several class libraries.

Java applet A small Java program designed to run within a Web browser. It is dowloadable and executable by a browser or network computer.

Java beans Java's component architecture, developed by Sun, IBM, and others. The components, called Java beans, can be parts of Java programs, or they can exist as self-contained applications. Java beans can be assembled to create complex applications, and they can run within other component architectures (such as ActiveX and OpenDoc).

Java Development Kit (JDK) The Java Development Kit is the set of Java technologies made available to licensed developers by Sun Microsystems. Each release of the JDK contains the following: the Java Compiler, Java Virtual Machine, Java Class Libraries, Java Applet Viewer, Java Debugger, and other tools.

Java Naming and Directory Interface (JNDI) A set of APIs that assist with the interfacing to multiple naming and directory services. (Definition copyright 1996-1999 Sun Microsystems, Inc. All Rights Reserved. Used by permission.)

Java Native Interface (JNI) A native programming interface that allows Java code running inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C and C++.

Java Platform The Java Virtual Machine and the Java Core classes make up the Java Platform. The Java Platform provides a uniform programming interface to a 100%. Pure Java program regardless of the underlying operating system. (Definition copyright 1996-1999 Sun Microsystems, Inc. All Rights Reserved. Used by permission.)

Java Remote Method Invocation (RMI) Java Remote Method Invocation is method invocation between peers, or between client and server, when applications at both ends of the invocation are written in Java. Included in JDK 1.1.

Java Runtime Environment (JRE) A subset of the Java Development Kit for end-users and developers who want to redistribute the JRE. The JRE consists of the Java Virtual Machine, the Java Core Classes, and supporting files. (Definition copyright 1996-1999 Sun Microsystems, Inc. All Rights Reserved. Used by permission.)

Java Servlet Servlets are similar to CGI programs, except that they are written in Java and run in a Java Virtual Machine managed by the Web server. Servlets are an effective substitute for CGI scripts because they provide an easier and faster way to generate dynamic documents. They also address the problem of doing server-side programming with platform-specific APIs because they are developed with the Java Servlet API, a standard Java extension. Servlets are modules that run inside Java-enabled Web servers and extend them in some manner. For example, a Servlet might be responsible for validating the data in an HTML order-entry form.

Java Virtual Machine (JVM) A software implementation of a central processing unit (CPU) that runs compiled Java code (applets and applications).

JavaDoc Sun's tool for generating HTML documentation on classes by extracting comments from the Java source code files.

JavaScript A scripting language used within an HTML page. Superficially similar to Java but JavaScript scripts appear as text within the HTML page. Java applets, on the other hand, are programs written in the Java language and are called from within HTML pages or run as stand-alone applications.

JDBC (Java Database Connectivity) In the JDK, the specification that defines an API that enables programs to access databases that comply with this standard.

JIT See *Just-In-Time Compiler.*

JNDI See *Java Naming and Directory Interface.*

JNI See *Java Native Interface.*

JRE See *Java Runtime Environment.*

Just-In-Time compiler (JIT) A platform-specific software compiler often contained within JVMs. JITs compile Java bytecodes on-the-fly into native machine instructions, thereby reducing the need for interpretation.

JVM See *Java Virtual Machine*.

Kerberos A network authentication protocol that is designed to provide strong authentication for client/server applications by using secret-key cryptography.

Large object (LOB) See LOB.

Link-edit To create a loadable computer program using a linkage editor.

Linker A computer program for creating load modules from one or more object modules or load modules by resolving cross references among the modules and, if necessary, adjusting addresses. In Java, the linker creates an executable from compiled classes.

Load module A program unit that is suitable for loading into main storage for execution. The output of a linkage editor.

LOB A sequence of bytes representing bit data, single-byte characters, double-byte characters, or a mixture of single and double-byte characters. A LOB can be up to 2GB -1 byte in length. See also BLOB, CLOB, and DBCLOB.

Method A fragment of Java code within a class that can be invoked and passed a set of parameters to perform a specific task.

Middle tier The hardware and software that resides between the client and the enterprise server resources and data. The software includes a Web server that receives requests from the client and invokes Java Servlets to process these requests. The client communicates with the Web server via industry standard protocols such as HTTP and IIOP.

Middleware A layer of software that sits between a database client and a database server, making it easier for clients to connect to heterogeneous databases.

Multithreading Multiple TCBs executing one copy of DB2 ODBC code concurrently (sharing a processor) or in parallel (on separate central processors).

MVS/ESA™ Multiple Virtual Storage/Enterprise Systems Architecture.

Native code Machine-dependent C code that can be invoked from Java. For multi-platform work, the native routines for each platform need to be implemented.

Null A special value that indicates the absence of information.

Object The principal building block of object-oriented programs. Objects are software programming modules. Each object is a programming unit consisting of related data and methods.

ODBC See Open Database Connectivity.

ODBC driver A dynamically-linked library (DLL) that implements ODBC function calls and interacts with a data source.

Open Database Connectivity (ODBC) A Microsoft database application programming interface (API) for C that allows access to database management systems by using callable SQL. ODBC does not require the use of an SQL preprocessor. In addition, ODBC provides an architecture that lets users add modules called database drivers that link the application to their choice of database management systems at run time. This means that applications no longer need to be directly linked to the modules of all the database management systems that are supported.

Package (1) In Java, a program element that contains classes and interfaces. (2) In DB2, a control structure produced during program preparation that is used to execute SQL statements.

Parameter marker A question mark (?) that appears in a statement string of a dynamic SQL statement. The question mark can appear where a host variable might appear if the statement string was a static SQL statement.

Persistence In object models, a condition that allows instances of classes to be stored externally, for example in a relational database.

Plan See application plan.

Plan name The name of an application plan.

Precompilation A processing of application programs containing SQL statements that takes place before compilation. SQL statements are replaced with statements that are recognized by the host language compiler. Output from this precompilation includes source code that can be submitted to the compiler and the database request module (DBRM) that is input to the bind process.

Prepare The first phase of a two-phase commit process in which all participants are requested to prepare for commit.

Prepared SQL statement A named object that is the executable form of an SQL statement that has been processed by the PREPARE statement.

Primary key A unique, non-null key that is part of the definition of a table. A table cannot be defined as a parent unless it has a unique key or primary key.

Process A program executing in its own address space, containing one or more threads.

Property An initial setting or characteristic of a bean, for example, a name, font, text, or positional characteristic.

RDBMS See *Relational database management system*.

Read stability (RS) An isolation level that locks only the rows that an application retrieves within a transaction. Read stability ensures that any qualifying row that is read during a transaction is not changed by other application processes until the transaction is completed, and that any row changed by another application process is not read until the change is committed by that process. Read stability allows more concurrency than repeatable read, and less than cursor stability.

Reference An object's address. In Java, objects are passed by reference rather than by value or by pointers.

Relational database management system (RDBMS). A relational database manager that operates consistently across supported IBM systems.

Remote Refers to any object maintained by a remote DB2 subsystem; that is, by a DB2 subsystem other than the local one. A remote view, for instance, is a view maintained by a remote DB2 subsystem. Contrast with local.

Remote Method Invocation (RMI) RMI is a specific instance of the more general term RPC. RMI allows objects to be distributed over the network; that is, a Java program running on one computer can call the methods of an object running on another computer. RMI and java.net are the only 100% pure Java APIs for controlling Java objects in remote systems.

Remote Object Instance Manager In Remote Method Invocation, a program that creates and manages instances of server beans through their associated server-side server proxies.

Remote Procedure Calls (RPC) RPC is a generic term referring to any of a series of protocols used to execute procedure calls or method calls across a network. RPC allows a program running on one computer to call the services of a program running on another computer.

Repeatable read (RR) An isolation level that locks all the rows in an application that are referenced within a transaction. When a program uses repeatable read protection, rows referenced by the program cannot be changed by other programs until the program ends the current transaction.

Requester Also application requester (AR). The source of a request to a remote RDBMS, the system that requests the data.

RMI (Remote Method Invocation) See *Remote Method Invocation.*

Rollback The process of restoring data changed by SQL statements to the state at its last commit point. All locks are freed. Contrast with commit.

RPC See *Remote Procedure Calls*.

RRSAF Recoverable Resource Manager Services attachment facility, which is a DB2 UDB for OS/390 subcomponent that uses OS/390 Transaction Management and Recoverable Resource Manager Services to coordinate resource commitment between DB2 UDB for OS/390 and all other resource managers that also use OS/390 RRS in an OS/390 system.

Runtime system The software environment where compiled programs run. Each Java runtime system includes an implementation of the Java Virtual Machine.

Sandbox A restricted environment, provided by the Web browser, in which Java applets run. The sandbox offers them services and prevents them from doing anything naughty, such as doing file I/O or talking to strangers (servers other than the one from which the applet was loaded). The analogy of applets to children led to calling the environment in which they run the "sandbox."

Scalar function An SQL operation that produces asingle value from another value and is expressed as a function name followed by a list of arguments enclosed in parentheses. See also column function.

Secure Socket Layer (SSL) SSL is a security protocol which allows communications between a browser and a server to be encrypted and secure. SSL prevents eavesdropping, tampering or message forgery on your Internet or intranet network.

Security Features in Java that prevent applets downloaded off the Web from deliberately or inadvertently doing damage. One such feature is the digital signature, which ensures that an applet came unmodified from a reputable source.

Serialization Turning an object into a stream, and back again.

Server The computer that hosts the Web page that contains an applet. The .class files that make up the applet, and the HTML files that reference the applet reside on the server. When someone on the Internet connects to a Web page that contains an applet, the server delivers the .class files over the Internet to the client that made the request. The server is also known as the originating host.

Server bean The bean that is distributed using RMI services and is deployed on a server.

Servlet See Java Servlet.

Single precision A floating-point number that contains 32 bits. See also double precision.

Sourced function A function that is implemented by another built-in or user-defined function already known to the database manager. This function can be a scalar function or a column (aggregating) function; it returns a single value from a set of values (for example, MAX or AVG). Contrast with external function and built-in function.

SQL Structured Query Language. A language used by database engines and servers for data acquisition and definition.

SQL authorization ID (SQL ID) The authorization ID that is used for checking dynamic SQL statements in some situations.

SSL See secure socket layer.

Static bind. A process by which SQL statements are bound after they have been precompiled. All static SQL statements are prepared for execution at the same time. Contrast with dynamic bind.

Static SQL SQL statements, embedded within a program, that are prepared during the program preparation process (before the program is executed). After being prepared, the SQL statement does not change (although values of host variables specified by the statement might change).

Stored procedure A user-written application program, that can be invoked through the use of the SQL CALL statement.

Structured Query Language (SQL) A standardized language for defining and manipulating data in a relational database.

Table A named data object consisting of a specific number of columns and some number of unordered rows. Synonymous with base table or temporary table.

Task control block (TCB) An MVS control block used to communicate information about tasks within an address space that are connected to DB2. An address space can support many task connections (as many as one per task), but only one address space connection. See address space connection.

TCB See *Task control block*.

TCP/IP See *Transmission Control Protocol based on IP.*

Telnet Telnet provides a virtual terminal facility that allows users of one computer to act as if they were using a terminal connected to another computer. The Telnet client program communicates with the Telnet daemon on the target system to provide the connection and session.

Temporary table A table created by the SQL CREATE GLOBAL TEMPORARY TABLE statement that is used to hold temporary data. Contrast with result table.

Thin client Thin client usually refers to a system that runs on a resource-constrained machine or that runs a small operating system. Thin clients don't require local system administration, and they execute Java applications delivered over the network.

Third tier The third tier, or back end, is the hardware and software that provides database and transactional services. These back-end services are accessed through connectors between the middle-tier Web server and the third-tier server. Though this conceptual model depicts the second and third tier as two separate machines, the NCF model supports a logical three-tier implementation in which the software on the middle and third tier are on the same box.

Thread A separate flow of control within a program.

Timestamp A seven-part value that consists of a date and time expressed in years, months, days, hours, minutes, seconds, and microseconds.

Trace A DB2 facility that provides the ability to monitor and collect DB2 monitoring, auditing, performance, accounting, statistics, and serviceability (global) data.

Transmission Control Protocol based on IP (1) A network communication protocol used by computer systems to exchange information across telecommunication links. (2) An Internet protocol that provides for the reliable delivery of streams of data from one host to another.

Type In WSAD, a generic term for a class or interface.

UDF User-defined function

UDT User-defined data type

Uncommitted read (UR) An isolation level that allows an application to access uncommitted changes of other transactions. The application does not lock other applications out of the row it is reading, unless the other application attempts to drop or alter the table.

Unicode A 16-bit international character set defined by ISO 10646. See also ASCII.

Uniform Resource Locator (URL) The unique address that tells a browser how to find a specific Web page or file.

URL See *Uniform Resource Locator.*

User-defined data Type (UDT) See distinct type.

User-defined function (UDF) A function defined to DB2 using the CREATE FUNCTION statement that can be referenced thereafter in SQL statements. A user-defined function can be either an external function or a sourced function. Contrast with built-in function.

Variable (1) An identifier that represents a data item whose value can be changed while the program is running. The values of a variable are restricted to a certain data type. (2)A data element that specifies a value that can be changed. A COBOL elementary data item is an example of a variable. Contrast with constant.

Virtual machine A software or hardware implementation of a central processing unit (CPU) that manages the resources of a machine and can run compiled code. See *Java Virtual Machine*.

Web See *World Wide Web*.

Web browser The Web uses a client/server processing model. The Web browser is the client component. Examples of Web browsers include Mosaic, Netscape Navigator, and Microsoft Internet Explorer. The Web browser is responsible for formatting and displaying information, interacting with the user, and invoking external functions, such as Telnet, or external viewers for data types that it does not directly support. Web browsers are fast becoming the universal client for the GUI workstation environment, in much the same way that the ability to emulate popular terminals such as the DEC VT100 or IBM 3270 allows connectivity and access to character-based applications on a wide variety of computers. Web browsers are available for all popular GUI workstation platforms and are inexpensive (often included with operating systems or related products for no additional charge.)

Web server Web servers are responsible for servicing requests for information from Web browsers. The information can be a file retrieved from the server¢ s local disk or generated by a program called by the server to perform a specific application function. Web servers are sometimes referred to as httpd servers or daemons. A number of Web servers are available for most platforms including most UNIX variants, OS/2® Warp, OS/390, and Windows NT. In addition, commercial Web servers that offer higher levels of vendor support and additional function are available. IBM has released the IBM Internet Connection Secure Server (ICSS) and its follow-on, the Domino™ Go Web Server (DGW), for the AIX, OS/2 Warp, Windows NT, and OS/390 platforms.

WebSphere WebSphere is the cornerstone of IBM's overall Web strategy, offering customers a comprehensive solution to build, deploy and manage e-business Web sites. The product line provides companies with an open, standards-based, Web server deployment platform and Web site development and management tools to help accelerate the process of moving to e-business.

World Wide Web A network of servers that contain programs and files. Many of the files contain hypertext links to other documents available through the network.

WWW See *World Wide Web*.

XML The Extensible Markup Language (XML) is an important new standard emerging for structured documents on the Web. XML extends HTML beyond a limited tag set and adapts SGML, making it easy for developers to write programs that process this markup and providing for a rich, more complex encoding of information. The importance of XML is indicated by support from companies such as Microsoft and Netscape.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 328.

- *Squeezing the Most Out of Dynamic SQL with DB2 for z/OS and OS/390,* SG24-6418
- *DB2 for OS/390 and z/OS Powering the World’s e-business Solutions*, SG24-6257
- *e-business Cookbook for z/OS Volume III: Java Development,* SG24-5980
- *Distributed Functions of DB2 for z/OS and OS/390*, SG24-6952
- *WebSphere Studio Application Developer Version 5 Programming Guide,* SG24-6957
- *S/390 File and Print Serving*, SG24-5330
- *DB2 for z/OS and WebSphere: The Perfect Couple*, SG24-6319

Other resources

These publications are also relevant as further information sources:

- *Application Programming Guide and Reference for Java™*, SC26-9932
- *Application Programming Guide and Reference FOR JAVA™ Version 8,* SC18-7414-**01**
- *DB2 UDB for OS/390 and z/OS Installation Guide Version 7*, GC26-9936
- *DB2 UDB for OS/390 and z/OS Application Programming and SQL Guide*, SC26-9933
- *New IBM Technology featuring Persistent Reusable Machines*, SC34-6034
- *DB2 Universal Database for OS/390 and z/OS Messages and Codes*, GC26-9940
- *DB2 Universal Database for OS/390 and z/OS SQL Reference*, SC26-9944
- *CICS TS for z/OS V2.2 Java Applications in CICS*, SC34-6000
- *New IBM Technology featuring Persistent Reusable Machines*, SC34-6034
- *Unix System Services Command Reference*, SA22-7802
- *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834
- *Assembling Java™ 2 Platform, Enterprise Edition (J2EE™) Applications*, SA22-7836
- *WebSphere Application Server V4.0.1 for z/OS and OS/390: Systems Management: User Interface*, SA22-7838
- John Ellis et al., *JDBC 3.0 Specification*, Sun Microsystems, October 2001
- Hatcher et al., *Java Development With Ant*, Manning Publications Company, 2002, ISBN 1930110588

Referenced Web sites

These Web sites are also relevant as further information sources:

- Download site for DB2 Visual Explain

  `http://www.ibm.com/software/data/db2/os390/db2ve/`

- JDBC 3.0 specification

  `http://java.sun.com/products/jdbc/download.html`

- The Samba homepage

  `http://www.samba.org/`

How to get IBM Redbooks

You can order hardcopy Redbooks, as well as view, download, or search for Redbooks at the following Web site:

ibm.com/redbooks

You can also download additional materials (code samples or diskette/CD-ROM images) from that site.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Index

Symbols

Numerics

A

B

C

IBM

Redbooks

DB2 for z/OS and OS/390: Ready for Java

DB2 for z/OS and OS/390: Ready for Java

Setting up your Java-DB2 environment

Easy-to-use examples, including using SQLJ with WSAD V5.1

Java-DB2 usage hints and tips

The earth is flat!
The earth is the center of the universe!
Men shall never fly!
Java will never work properly on the mainframe!

All four statements had a lot of advocates for a long time, but all of them turned out to be wrong.

In this IBM Redbook we show how Java and DB2 for z/OS and OS/390 can work together and form a strong combination that can run your mission-critical enterprise applications. This publication focusses on the new IBM Universal Driver for SQLJ and JDBC, IBM's new JDBC driver implementation, supporting both Type 2 and Type 4 driver connectivity to the members of the DB2 family, including DB2 for z/OS, and DB2 for Linux, Unix and Windows.

This publication provides guidance on the different ways to set up your environment to hook a Java program up to a DB2 for z/OS subsystem, through JDBC or SQLJ, using the Type 2 driver and the Type 4 driver. We provide an SQLJ tutorial, and demonstrate how to develop and deploy SQLJ programs using the new SQLJ support functions that became available with WebSphere Studio Application Developer V5.1. We demonstrate the use of Java and DB2 using native Java programs, as well as through the use of Servlets and JSPs running on a WebSphere Application Server.

SG24-6435-00 ISBN 0738427853