



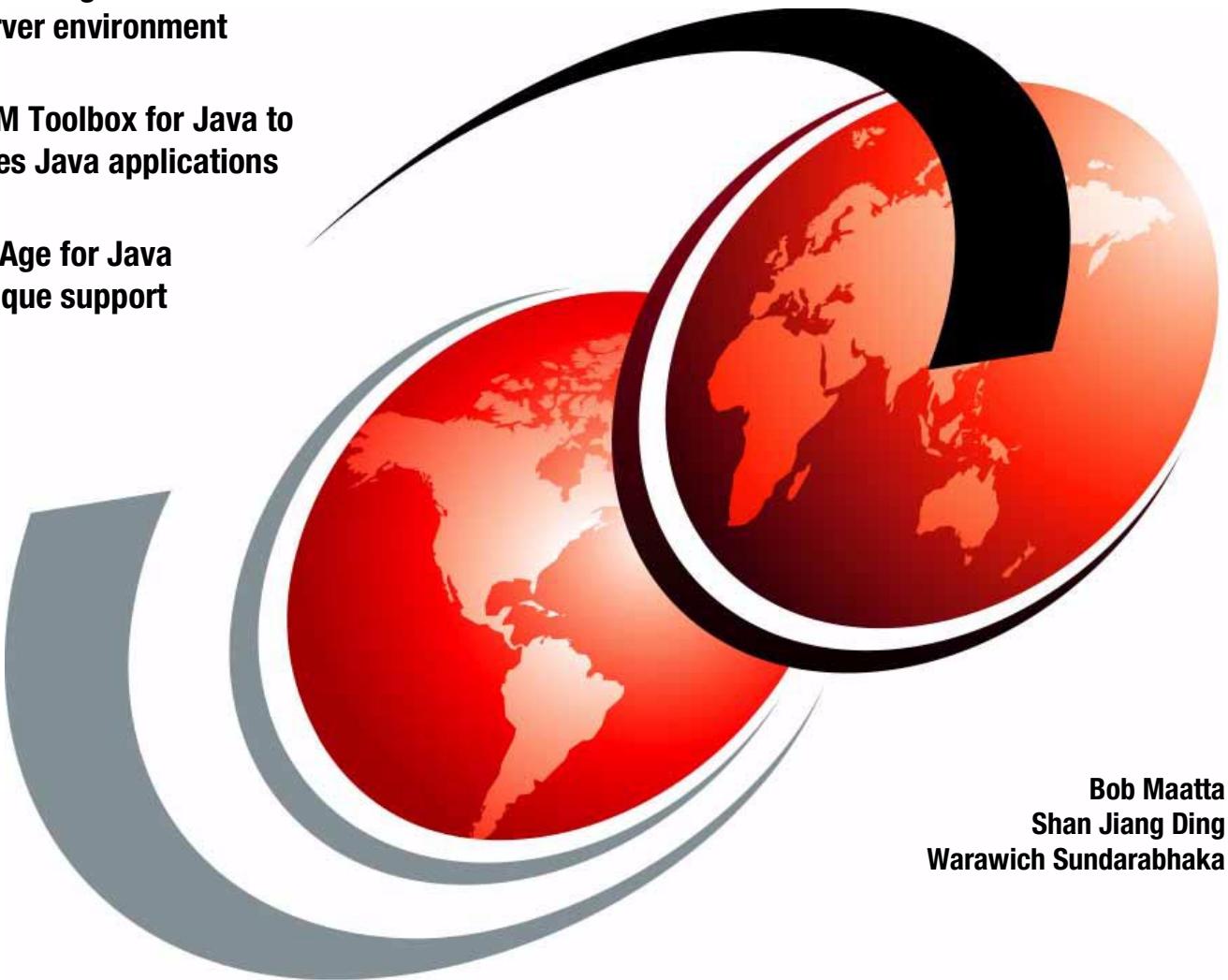
# Building Java Applications for the iSeries Server

## with VisualAge for Java 3.5

Install and configure Java for the  
iSeries server environment

Use the IBM Toolbox for Java to  
build iSeries Java applications

Use VisualAge for Java  
iSeries unique support



Bob Maatta  
Shan Jiang Ding  
Warawich Sundarabhaka





International Technical Support Organization

**Building Java Applications for the iSeries Server  
with VisualAge for Java 3.5**

June 2001

**Take Note!** Before using this information and the product it supports, be sure to read the general information in "Special notices" on page 531.

**First Edition (June 2001)**

This edition applies to V5R1 of OS/400.

This document created or updated on January 31, 2002.

Comments may be addressed to:  
IBM Corporation, International Technical Support Organization  
Dept. JLU Building 107-2  
3605 Highway 52N  
Rochester, Minnesota 55901-7829

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

**© Copyright International Business Machines Corporation 2001. All rights reserved.**

Note to U.S Government Users - Documentation related to restricted rights - Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.





# Contents

<b>Contents .....</b>	v
<b>Preface .....</b>	xi
The team that wrote this Redbook .....	xi
Special notice .....	xiii
IBM Trademarks .....	xiii
Comments welcome .....	xiii
<b>Chapter 1. Java overview and iSeries implementation .....</b>	1
1.1 Java as a programming language .....	2
1.1.1 Before object-oriented technology .....	2
1.1.2 Objects .....	3
1.1.3 Class relationships .....	4
1.1.4 Polymorphism .....	6
1.1.5 Benefits of object-oriented technology .....	6
1.2 Java overview and iSeries implementation .....	8
1.3 Java platform .....	8
1.3.1 Java virtual machine .....	9
1.3.2 Java APIs .....	11
1.3.3 Java utilities .....	13
1.4 Java on the iSeries server .....	16
1.4.1 iSeries Java virtual machine .....	17
1.4.2 Java APIs and the iSeries server .....	19
1.4.3 Java utilities and the iSeries server .....	19
1.5 iSeries-specific implementation .....	22
1.5.1 The OS/400 Java commands .....	23
1.5.2 The Qshell Interpreter .....	32
1.5.3 Remote AWT support .....	34
<b>Chapter 2. iSeries Java configuration .....</b>	37
2.1 Checking the required software on the iSeries server .....	38
2.1.1 Setting up the Java version to use .....	44
2.2 Setting up the environment on the iSeries server .....	44
2.2.1 Setting up the Java environment for CL commands .....	45
2.2.2 Setting up the environment for the Qshell Interpreter .....	47
2.2.3 iSeries CLASSPATH recommendation .....	50
2.2.4 Testing the Java environment on the iSeries server .....	50
2.2.5 Setting the QUTCOFFSET system value .....	52
2.3 Installing the IBM Toolbox for Java on the workstation .....	53
2.3.1 Setting up the CLASSPATH variable .....	54
2.3.2 Copying the IBM Toolbox for Java classes to the workstation .....	56
2.4 Using Remote AWT support on the workstation .....	57
2.4.1 Setting up the Remote AWT environment .....	57
2.4.2 Starting Remote AWT support on the workstation .....	58
2.4.3 Running Remote AWT support on the iSeries server .....	59
<b>Chapter 3. Introduction to VisualAge for Java .....</b>	63
3.1 VisualAge for Java overview .....	64
3.1.1 VisualAge for Java versions .....	64

3.1.2 Integrated Development Environment . . . . .	65
3.2 Using the Integrated Development Environment . . . . .	66
3.2.1 How it fits together . . . . .	78
3.2.2 Building a sample application . . . . .	82
3.2.3 Team development . . . . .	106
3.2.4 Applets and Applet Viewer . . . . .	110
3.2.5 Editor, debugger, and SmartGuides . . . . .	112
3.3 System requirements and prerequisites for version 3.5 . . . . .	119
3.4 Summary . . . . .	120
<b>Chapter 4. IBM Toolbox for Java . . . . .</b>	<b>123</b>
4.1 Introduction to the IBM Toolbox for Java . . . . .	124
4.1.1 Installing the Toolbox . . . . .	124
4.2 Enhancements . . . . .	127
4.2.1 V4R3 enhancements . . . . .	127
4.2.2 V4R4 enhancements . . . . .	128
4.2.3 V4R5 enhancements . . . . .	130
4.2.4 V5R1 enhancements . . . . .	131
4.3 IBM Toolbox for Java and host servers. . . . .	132
4.3.1 AS400 object, infrastructure, and sign-on. . . . .	133
4.4 The base API package . . . . .	134
4.4.1 Data descriptions and conversions . . . . .	135
4.4.2 iSeries data types . . . . .	135
4.4.3 Record-level conversions . . . . .	136
4.4.4 JDBC specification . . . . .	137
4.4.5 Record-level access . . . . .	145
4.4.6 Integrated file system . . . . .	145
4.4.7 Print. . . . .	146
4.4.8 Command call . . . . .	146
4.4.9 Program call . . . . .	147
4.4.10 Data queue . . . . .	147
4.4.11 Proxy support . . . . .	148
4.4.12 Servlet support . . . . .	148
4.5 How the iSeries server fits into this picture . . . . .	149
4.5.1 Security . . . . .	150
4.5.2 National language support . . . . .	150
4.5.3 Save and restore considerations . . . . .	151
4.5.4 Error recovery considerations . . . . .	151
4.5.5 Mapping iSeries data types to Java data types. . . . .	151
4.6 Using the access classes . . . . .	152
4.6.1 iSeries database access . . . . .	152
4.6.2 JDBC application example . . . . .	153
4.6.3 JDBC 2.0. . . . .	163
4.6.4 JDBC 2.0 example . . . . .	167
4.6.5 Reusable GUI part . . . . .	170
4.6.6 Stored procedures . . . . .	171
4.6.7 JDBC stored procedure application example . . . . .	172
4.6.8 DDM record-level access application example . . . . .	179
4.6.9 Distributed Program Call feature. . . . .	185
4.6.10 Distributed Program Call (DPC) application example . . . . .	186
4.6.11 Data queues . . . . .	194
4.6.12 Data queue application example. . . . .	196
4.6.13 Network print. . . . .	206

4.6.14 Print example .....	206
4.6.15 Integrated file system access .....	210
4.6.16 Integrated file system example .....	211
4.6.17 Additional access classes .....	216
4.7 Introduction to GUI component classes .....	219
4.7.1 Overview of the GUI classes.....	219
4.7.2 JDBC examples .....	226
4.7.3 SQLResultSetFormPane .....	233
4.7.4 Record-level access GUI examples .....	238
4.7.5 Additional GUI component classes.....	242
4.8 Introduction to proxy support.....	252
4.8.1 Classes enabled to work with proxy server.....	253
4.8.2 Proxy support example .....	253
4.9 Conclusion .....	256
<b>Chapter 5. IBM Enterprise Toolkit for AS/400.....</b>	<b>257</b>
5.1 Using ET/400 .....	258
5.2 IBM Toolbox for Java classes .....	258
5.3 Distributed Program Call SmartGuide.....	259
5.3.1 Distributed Program Call feature.....	259
5.3.2 Application description .....	259
5.3.3 Creating a Program Call JavaBean.....	261
5.3.4 Building an application using the DPCXRPG bean in the VCE.....	265
5.4 SmartGuide to convert iSeries display files to Java .....	268
5.5 iSeries beans .....	271
5.5.1 JFormatted beans .....	271
5.5.2 Data File Utility (DFU) beans .....	279
5.6 Support for export, compile, run iSeries programs .....	293
5.6.1 Setup.....	293
5.6.2 Export support.....	295
5.6.3 Compile support .....	295
5.6.4 Run support .....	296
5.7 ET/400 system requirements .....	297
<b>Chapter 6. Overview of the Order Entry application .....</b>	<b>299</b>
6.1 Overview of the Order Entry application .....	300
6.1.1 The ABC Company .....	300
6.1.2 The ABC Company database .....	300
6.1.3 A customer transaction .....	300
6.1.4 Application flow .....	301
6.1.5 Customer transaction flow.....	302
6.1.6 Database table structure .....	307
6.1.7 Order Entry application database layout.....	307
6.1.8 Database terminology .....	311
<b>Chapter 7. Java for RPG programmers .....</b>	<b>313</b>
7.1 Object-oriented programming and RPG .....	314
7.1.1 What is Java .....	315
7.1.2 Java syntax.....	316
7.1.3 Object creation .....	317
7.1.4 Class variables .....	317
7.1.5 Class methods .....	317
7.1.6 Instance variables .....	318
7.1.7 Instance methods .....	318

7.1.8 Thread.....	318
7.1.9 Object destruction.....	318
7.1.10 Subclasses and inheritance .....	318
7.1.11 Overloading and overriding methods .....	318
7.1.12 Compiling Java on the iSeries server .....	319
<b>Chapter 8. Migrating the user interface to the Java client.....</b>	<b>321</b>
8.1 Re-designing the application.....	322
8.2 Creating the Java client graphical user interface .....	323
8.3 Overview of the Parts Order Entry window .....	323
8.4 Application flow through the Java client Order Entry window .....	324
8.4.1 Connecting to the database .....	325
8.4.2 Program interfaces .....	328
8.4.3 Retrieving the customer list.....	328
8.4.4 Retrieving the item list.....	333
8.4.5 Verifying and adding the item to the order .....	337
8.4.6 Submitting the order .....	338
8.5 Changes to the host Order Entry application .....	344
8.5.1 Providing a customer list.....	344
8.5.2 Providing an item list.....	345
8.5.3 Verifying an item .....	346
8.5.4 Processing the submitted order .....	346
8.6 Summary.....	348
<b>Chapter 9. Moving the server application to Java .....</b>	<b>349</b>
9.1 What RMI is.....	350
9.2 Building an RMI application.....	351
9.3 Building a simple iSeries application using RMI .....	352
9.3.1 Defining interfaces .....	352
9.3.2 Implementing the remote server object.....	352
9.3.3 Running rmic on a remote implementation class .....	353
9.3.4 Implementing the client.....	354
9.3.5 Making the server code network accessible .....	355
9.4 RMI JDBC example.....	355
9.4.1 Item class .....	358
9.4.2 Defining the interface .....	359
9.4.3 Implementing the remote server object.....	359
9.4.4 Creating the stubs and skeletons .....	363
9.4.5 Implementing the client .....	365
9.4.6 Making the server code network accessible .....	368
9.5 Moving the Order Entry server application to Java .....	370
9.6 Order Entry using record-level access (DDM).....	373
9.6.1 Method logic .....	376
9.6.2 Cleaning up.....	386
9.7 Order Entry using JDBC .....	388
9.7.1 Method logic .....	390
9.7.2 Cleaning up.....	397
9.8 Remote method invocation support .....	397
9.8.1 RMI application design .....	398
9.8.2 Adding RMI support to a server class.....	399
9.8.3 Adding RMI support to the client.....	400
9.8.4 Creating a client class to handle RMI .....	401
9.9 Conclusion .....	403

<b>Chapter 10. Structured Query Language for Java (SQLJ) . . . . .</b>	405
10.1 Using SQLJ . . . . .	406
10.2 Introduction to SQLJ . . . . .	406
10.2.1 An overview of how SQLJ works . . . . .	407
10.2.2 SQLJ and the iSeries server . . . . .	408
10.2.3 Additional information . . . . .	408
10.3 The Cstmrlnq example . . . . .	408
10.3.1 Setting up VisualAge for Java for the Cstmrlnq example . . . . .	408
10.3.2 Running Cstmrlnq . . . . .	411
10.3.3 Cstmrlnq explanation . . . . .	414
10.4 The CstmrList example . . . . .	417
10.4.1 Setting up VisualAge for Java for the CstmrList example . . . . .	417
10.4.2 Running CstmrList . . . . .	417
10.4.3 CstmrList explanation . . . . .	418
10.5 Order Entry client application . . . . .	421
10.5.1 Converting the OrderEntryWdw class to use SQLJ . . . . .	423
10.5.2 Converting SltCustWdw to use SQLJ . . . . .	424
10.5.3 Converting SltItemWdw to use SQLJ . . . . .	427
10.6 Order Entry server application . . . . .	428
10.6.1 Setting up the Order Entry server application . . . . .	429
10.6.2 Converting the Order Entry to use SQLJ . . . . .	430
10.7 SQLJ considerations . . . . .	437
<b>Chapter 11. Java Native Interface (JNI) . . . . .</b>	439
11.1 Introduction to Java Native Interface . . . . .	440
11.1.1 JNI positioning . . . . .	440
11.1.2 Who should use JNI . . . . .	441
11.1.3 Additional information . . . . .	441
11.2 Java Native Interface and RPG . . . . .	442
11.2.1 RPG native method example . . . . .	443
11.2.2 Calling a Java method from the RPG example . . . . .	446
11.2.3 RPG and JNI consideration . . . . .	450
11.3 RPG Order Entry example . . . . .	450
11.3.1 Converting the Order Entry application to use JNI . . . . .	451
11.3.2 The RPG service program . . . . .	455
11.4 Java Native Interface and C . . . . .	464
11.4.1 Setting up JNI and C . . . . .	464
11.4.2 Hello C example . . . . .	466
11.4.3 Changing a Java String object from a C program . . . . .	468
11.4.4 C with the Java Invocation API . . . . .	471
11.4.5 C program: INVOKEJAVA . . . . .	472
<b>Chapter 12. Debugging Java programs on the iSeries server . . . . .</b>	477
12.1 Getting ready to debug . . . . .	478
12.1.1 Compiling the code for debugging . . . . .	478
12.2 Using the OS/400 system debugger . . . . .	479
12.2.1 Setting breakpoints . . . . .	480
12.2.2 Displaying variables . . . . .	482
12.2.3 Work with module list . . . . .	485
12.2.4 Debugging from another terminal session . . . . .	486
12.3 The IBM Distributed Debugger . . . . .	489
12.3.1 Starting the Distributed Debugger . . . . .	489
12.3.2 Debugging an iSeries Java program . . . . .	492
12.3.3 Controlling the Distributed Debugger session . . . . .	495

<b>Chapter 13. Deployment considerations and tools</b>	497
13.1 The IBM Toolbox for Java installation and update	498
13.1.1 Embedding the AS400ToolboxInstaller class in your program	498
13.1.2 Running the AS400ToolboxInstaller class from the command line	499
13.2 Java archive files	500
13.2.1 JarMaker	500
13.2.2 JarMaker example	502
13.2.3 AS400ToolboxJarMaker	503
13.2.4 Example usage	506
13.3 Securing applications with SSL	506
13.3.1 Internet security elements	507
13.3.2 Transaction security and Secure Sockets Layer	508
13.4 Digital certificates and certificate authority	511
13.5 iSeries implementation of Digital Certificate Management	512
13.5.1 Changing the authority of the directory	512
13.5.2 Configuring a digital certificate environment	513
13.6 Using a self-signed certificate for SSL	513
13.6.1 Creating an intranet certificate authority	513
13.6.2 Creating a server certificate with your intranet CA	517
13.7 Using a server certificate from an Internet CA	518
13.7.1 Receiving a server certificate for this server	521
13.8 Using a certificate with the IBM Toolbox for Java	522
13.8.1 Using a certificate from a trusted authority	522
13.8.2 Using a self-signed certificate	522
13.9 Modifying an application to use SSL with VisualAge 3.5	524
13.9.1 Importing the required classes	525
13.9.2 Modifying the program	526
13.9.3 Testing the changed program	527
13.9.4 Additional SSL-related resources	528
<b>Appendix A. Additional material</b>	529
Locating the Web material	529
Using the Web material	529
<b>Special notices</b>	531
<b>Related publications</b>	533
IBM Redbooks	533
Referenced Web sites	534
How to get IBM Redbooks	534
<b>List of Abbreviations</b>	537
<b>Index</b>	539

# Preface

In the past several years, Java has become the hot new programming language. The reasons for Java's popularity are its portability, robustness, and ability to produce Internet-enabled applications.

This IBM Redbook explains how you can use Java and the IBM @server iSeries server to build server applications and client/server applications for the new network computing paradigm. It focuses on two key products: VisualAge for Java Version 3.5 and IBM Toolbox for Java.

Throughout this Redbook, you'll find many practical programming examples with detailed explanations on how they work. You'll see how to modernize legacy RPG applications in a practical and evolutionary way through client and server Java examples. These examples are available for download from the Redbooks Web site. To understand this code better, download the files, as explained in Appendix A, and use them as a reference.

This Redbook is intended to help customers and service providers who need to install and configure Java on the iSeries server. It also targets application developers who want to develop Java applications for the iSeries server. Reading this Redbook will help you to quickly and easily start using Java with the iSeries server.

**Note:** This Redbook was developed using OS/400 V4R5 and V5R1. It replaces the Redbooks *Building AS/400 Client/Server Applications with Java*, SG24-2152, and *Building AS/400 Applications with Java*, SG24-2163.

## 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 Rochester Center.

**Bob Maatta** is a Senior Software Engineer from the United States at the IBM International Technical Support Organization, Rochester Location. He is the ITSO technical leader for iSeries e-business application development. He writes extensively and teaches IBM classes worldwide on all areas of iSeries Java and e-business application development. Before joining the ITSO in 1995, he worked in the AS/400 National Technical Support Center as a Consulting Market Support Specialist. He has over 20 years of experience in the computer industry. He has worked on numerous computer platforms including S/390, S/38, AS/400, and Personal Computers. He is a Sun Certified Java Programmer and a Sun Certified Java Developer.

**Shan Jiang Ding** is an Advisory I/T Specialist with the IBM Technical Support Centre in China. He worked on several iSeries client/server projects before joining IBM three years ago. His areas of expertise include WebSphere, Java, and Client Access. He is a Sun Certified Java Programmer.

**Warawich Sundarabhaka** is an Advisory I/T Specialist with IBM Global Services in Thailand. He has been with IBM since 1991. He has over 20 years of experience in the computer field and has worked with the AS/400 system since 1988. His areas of iSeries expertise include performance management, application development using Java, RPG, COBOL, and DB2 UDB for iSeries. He has taught iSeries courses for IBM Thailand education.

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

Hal Frye  
**IBM Boulder**

Jim Fair  
Jeff Lee  
Leonardo Llames  
Jennifer Maynard  
Cheryl Renner  
Kevin Roberts  
Chris Smith  
Dave Wall  
Robb Wiedrich  
Neil Willis  
Blair Wyman  
**IBM Rochester**

Pierre Goudet  
**IBM France**

Roger Wong  
**IBM Hong Kong**

Mohammad Omar Nishtar  
**IBM Pakistan**

Vadim Berestetsky  
Barbara Morris  
**IBM Toronto**

Stuart Foster  
Dan Murphy  
**IBM UK**

Markus Abegglen  
Daniel Stucki  
**DV Bern AG**

Simon Coulter  
**FlyByNight Software**

Marshall Dunbar  
**Data Processing Services, Inc.**

Paul Holm  
**Planet-J**

Craig Pelkie  
**Bits&Bytes Programming**

Brian Skaarup  
**EDB Gruppen Systems A/S**

## Special notice

This publication is intended to help anyone who wants to use Java on an iSeries server. The information in this publication is not intended as the specification of any programming interfaces that are provided by the IBM Toolbox for Java or the iSeries Developer Kit for Java products.

## IBM Trademarks

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

e (logo)® 	Redbooks
IBM ®	Redbooks Logo 
Advanced Function Printing	RPG/400
AFP	S/390
AIX	SecureWay
AS/400	Service Director
AT	SP
Common User Access	System/36
CT	System/38
Current	TeamConnection
DB2	VisualAge
FFST	WebSphere
First Failure Support Technology	World Registry
GDDM	XT
Integrated Language Environment	400
Language Environment	Lotus
Network Station	Domino
Operating System/400	Tivoli
OS/2	TME
OS/400	

## Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Please 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](http://ibm.com/redbooks)
- ▶ Send your comments in an Internet note to:  
[redbook@us.ibm.com](mailto:redbook@us.ibm.com)
- ▶ Mail your comments to the address on Page ii.





# Java overview and iSeries implementation

Java is an object-oriented programming language that is platform independent. It has become very popular as the programming language of the Internet. Once you learn how to program in Java, you can run your programs on any platform that is Java enabled. The iSeries is such a platform.

You can use Java to build client/server applications that use Java to access iSeries resources from a client platform or server-side Java applications that actually run Java code on the iSeries server. This chapter introduces Java as an object-oriented programming language and explains how it is implemented on the iSeries server.

# 1.1 Java as a programming language

Java is an object-oriented language. This chapter reviews object-oriented principles, but does not explain them in great detail. For a full introduction to object technology, refer to *Object-Oriented Technology: A Managers Guide*, SH20-9092, which is one of the best books on the subject.

This section covers information on:

- ▶ Objects
- ▶ Classes
- ▶ Class relationships
- ▶ Polymorphism
- ▶ Benefits of object-oriented technology

## 1.1.1 Before object-oriented technology

Staying competitive in the business world means seeking a better, more reliable software technology that actually delivers on its claims. The advent of object technology has done just that. It has rapidly closed the gap between hardware potential and software performance. As computers continue to gain in speed and power, the implementation of object-oriented technology becomes increasingly important.

Let us take a moment to review the traditional application development scenario. Do you recognize the scenario in Figure 1-1?

In this scenario, our applications were designed about 20 years ago to segregate the procedures from the data. They did this by using techniques such as information engineering (to normalize our databases) and functional decomposition to split the functions down into manageable chunks of code. Rarely, if ever, did we try to think of our small normalized database tables and our small code modules/programs/subroutines as entities that benefit more by being designed together.

On day one, the application was perfect. The modules were small and discrete, and our data was well normalized with clear and well-defined links between the modules and the data. Three months passed and the users loved the application, but then the first request came. This request was to extend the application a little. And, the second request was to fix a small bug that had appeared. Maybe we were lucky this time. The impact on the total application was simply to make a couple of minor modifications to the code modules and to add a couple of extra links from a module to the database. However, they were not in the original design.

Suppose this scenario continues for the next 20 years, with a couple of changes coming in every two to three months. Even the best application programmer/application developer (AP/AD) professional has great difficulty in retaining anything similar to the original design. Given that our programmers and designers have moved on two or three times from the original team, it is easy to see how the next picture in this scenario has evolved. Do you recognize the picture in Figure 1-2? Is this your application?

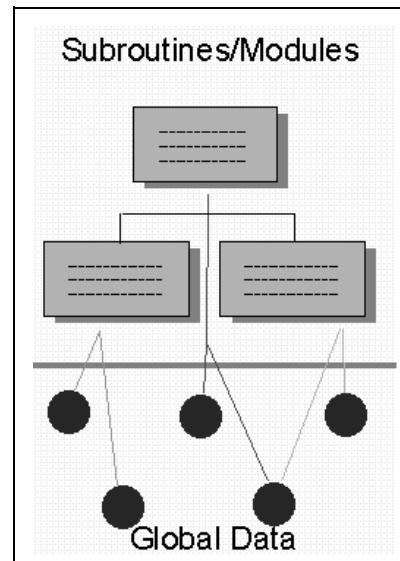


Figure 1-1 Traditional application development scenario

But wait a minute. From 20 years ago, we moved from 2GL to 3GL, 3GL to 4GL, 4GL to case, and case to uppercase and lowercase. Each of these transitions has made an incrementally better impact on software quality and design, and on programmer productivity. As an industry, we are still left trying to maintain these creaking systems, add real value to business, provide a competitive advantage with Web-based applications, and so on. The industry has been looking for a new way to develop applications that simulate the real world better. The industry does not do this by splitting data and functions apart and meshing them back together again as we have done. Instead, it does this by keeping the data and procedures together from analysis, through design, all the way to coding. This way of building systems is the object-oriented way. Let us see what this actually means.

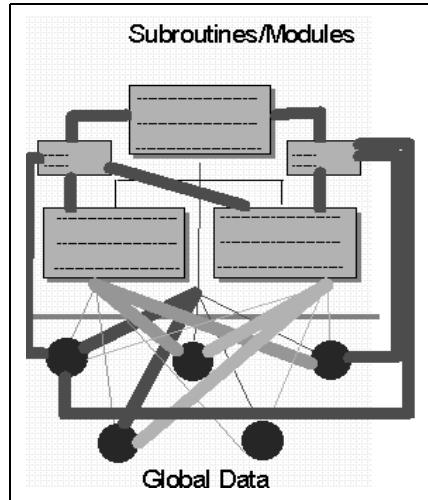


Figure 1-2 Updated application development scenario

### 1.1.2 Objects

An *object* is a software package that contains a collection of related procedures and data. In the object-oriented approach, procedures go by the name methods/member functions. In keeping with traditional programming terminology, the data elements are referred to as *variables/member variables/data members* because their values change over time.

#### Encapsulation of objects

The act of grouping both data and the operations that affect that data into a single object is known as *encapsulation*. Encapsulation is a powerful technique for building better software because it provides neat, manageable units that can be developed, tested, and maintained independently of one another. The knowledge encapsulated within an object can be hidden from external view. Consequently, the knowledge encapsulated within an object looks different from outside the object than it does within it. As with each of us, objects have a private side. The private side of an object is how it performs actions, and it can do them in any way that is required. How it performs the operations or computes the information is not a concern of other parts of the system. Using this principle, known as *information hiding*, objects are free to change their private sides without affecting the entire system.

Objects that share the same behavior are said to belong to the same class. A *class* is a generic specification for an arbitrary number of similar objects (see Figure 1-3). Objects that behave in a manner specified by a class are called *instances* of that class. All objects are instances of some class. Once an instance of a class is created, it behaves the same as all other instances of its class. Upon receiving a message, it can perform any operation for which it has methods. It may also call on other instances, of the same or other

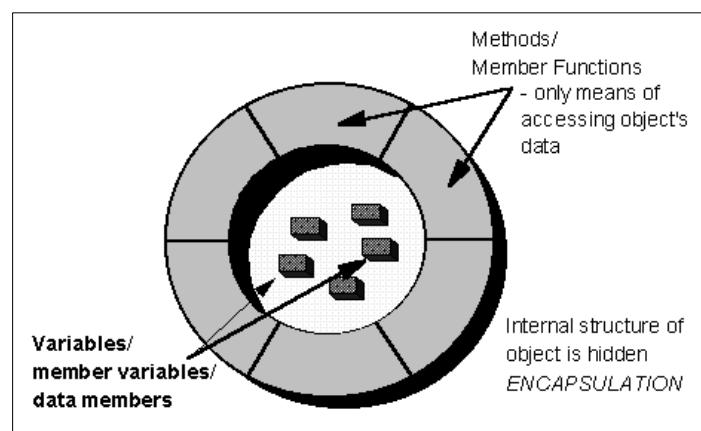


Figure 1-3 Classes

classes, to perform other operations on its behalf. A program can contain as many or as few instances of a particular class as required.

In theory, a class is a template for objects. Once the template is defined, it can stamp out as many objects (instances of the class) as desired. Each can take on different values, but all use the same variables and work with the same methods. This is how you can have a thousand different product objects but define the method for computing the price in only one place. See Figure 1-4.

To conclude, note these points:

- ▶ A class is a template that defines the methods and variables to be included in a particular type of object.
- ▶ The descriptions of the methods and variables that support them are defined only once in the definition of the class.
- ▶ The objects that belong to a class, called instances of the class, contain only their particular values for the variables.

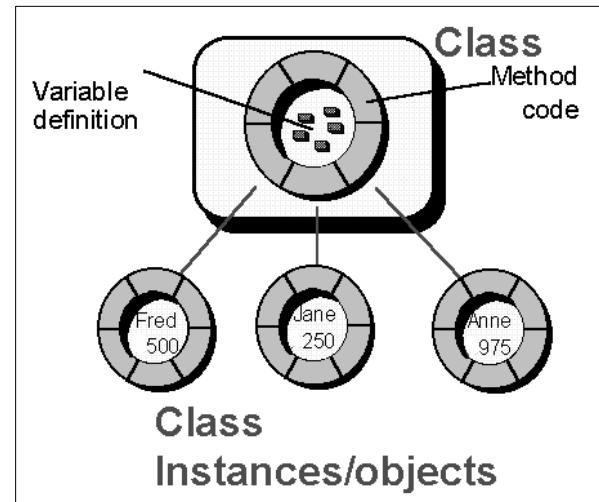


Figure 1-4 Instantiating objects

### 1.1.3 Class relationships

It is important to understand the relationship among classes. There are only three ways that classes can be connected together:

- ▶ Specialization
- ▶ Composition
- ▶ Collaboration

#### Specialization

By declaring one class to be a special case, or a subclass of another, the subclass inherits all the method and variable definitions of its super class. In the class hierarchy shown in Figure 1-5, *vehicle* is the super class of all the other subclasses, and *car* is a subclass of *vehicle* (because it is a type of vehicle). *Car* is the super class of its four subclasses

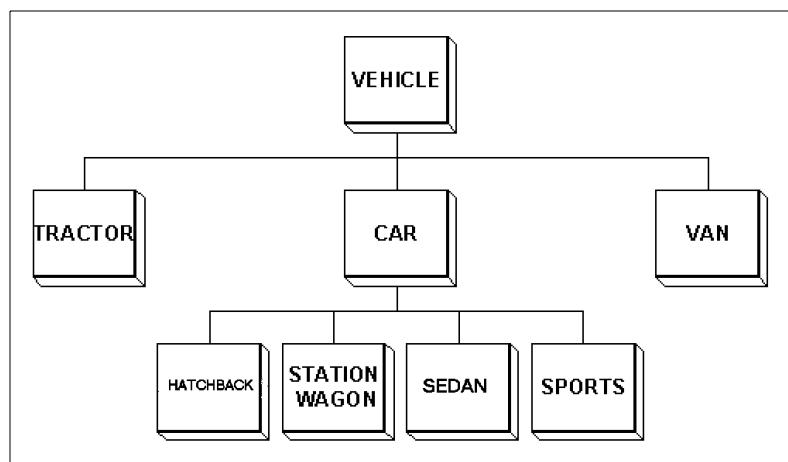


Figure 1-5 Class hierarchy

(Hatchback, Station

Wagon, Sedan, and Sports). These last four classes are subclasses of *car* because they are a type of car.

It is good to arrange classes into a hierarchy, but what features does the hierarchy have and what benefits does this bring? The vehicle class abstracts as much data and procedures that are common to all vehicle types. It also implements these data items (variables) and functions.

As shown in Figure 1-6, the vehicle class defines the regno variable (registration number or plate number) and all the functions that act on regno, for example, to set its value and to retrieve it (commonly called *setters* and *getters*). These are defined only once at the vehicle class level, but they are immediately inherited by the seven subclasses shown in this hierarchy.

There is no copying and pasting of code and no retying. It all happens automatically. This has a dramatic effect on the amount of code that needs to be written, on the quality of the code, and on the downstream maintenance effort (because you amend it in one place only, not in eight places). Therefore, in Figure 1-6, the sedan class has a variable and methods for trunk capacity (which it defines itself), for trim (which it inherits from car), and for regno (which it inherits from vehicle).

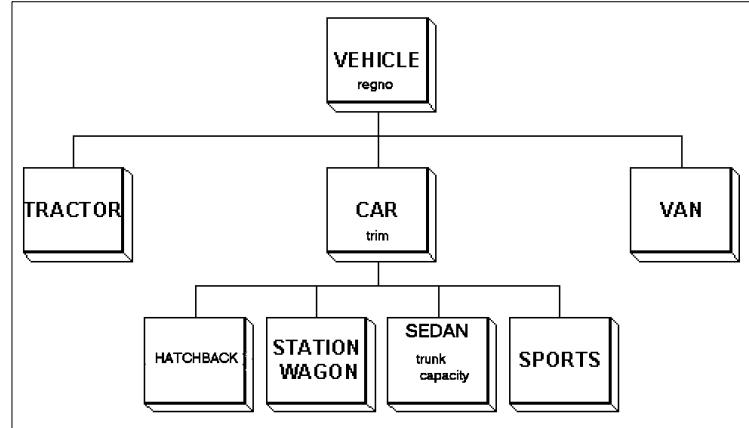


Figure 1-6 Inheritance

## Composition

Classes can also be defined as components of one another. A laser printer may contain, among many other parts, a print engine, a roller, a cartridge, a paper tray, and so on. Composition provides a convenient means of capturing the fact that these parts all go together, and it allows them to be treated as a single collective entity. Composition is especially useful for defining high-level objects that hide the details of their inner workings.

A division may consist of a specified set of departments, several divisions can be combined into a business unit, and a company may include any number of business units. It is important not to confuse specialization with composition. They have different properties and serve different functions. For example, the hierarchy defined by an organization is not an inheritance hierarchy. Departments do not inherit properties from divisions, and divisions do not inherit from business units. That is because they are components of one another; they are not special cases of each other.

## Collaboration

The final class relationship is one that triggers objects into action. A collaboration between two objects is a request from one object to another to carry out one of its services. The request takes the form of a message from the first object, called the *sender*, to the second object, called the *receiver*. The message consists of the name of a method defined by the receiver together with any information (expressed as parameters or arguments) that the receiver needs to carry out that method.

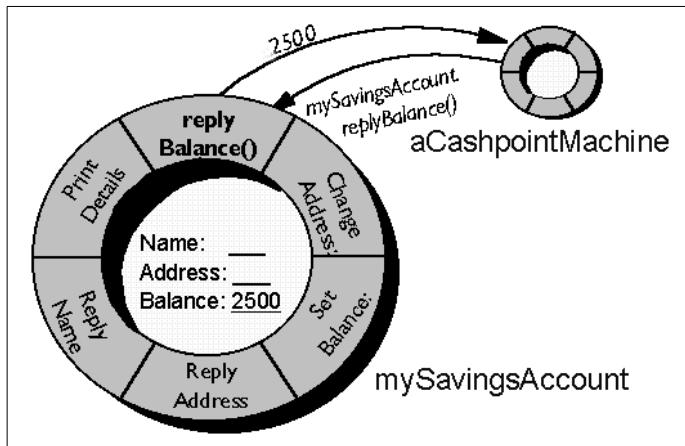


Figure 1-7 Collaboration

Collaborations, as demonstrated in Figure 1-7, provide the active element in object technology. The other two relationships, important as they are, are merely packaging rules that define how objects are composed. It is the passing of messages among objects during the execution of a program that actually makes the objects carry out their tasks.

#### 1.1.4 Polymorphism

Understanding how object-oriented software works leads to realizing its vast benefits. One of the most important benefits is an abstraction known as *polymorphism*. Simply put, polymorphism is the ability of two or more classes of an object to respond to the same message, each in its own way. This means that an object does not need to know to whom it is sending a message. It only needs to know that many different kinds of objects are defined to respond to that particular message. The concern is sending the right message. It is up to the receiver to interpret the request and do the correct thing.

Closely related to polymorphism is the concept of *dynamic binding*. This idea stresses that because the sender of a message does not know anything about its receiver, determining the identity of that receiver can be left until the program is actually running. The advantage of dynamic binding is that it leaves all of your options open until the moment the message is actually sent. In fact, fundamental changes can be made in the way a system works by simply adding new kinds of objects, without recompiling any programs or modifying existing classes.

#### 1.1.5 Benefits of object-oriented technology

Since object technology delivers speed improvements, it is important to recognize from where that added speed comes. Merely programming with objects is not faster than other kinds of programming. The increased speed does not come from programming faster, but from programming less. The critical factor is to build up an inventory of reusable class definitions so that new applications can be constructed largely by recombining existing classes. The more reuse that is implemented, the greater the benefit is.

Encapsulation allows the building of entities that can be depended on to behave in certain ways and know certain information. Such entities can be reused in every application that can use this behavior and knowledge. While it is possible to construct entities that are useful in many situations, using object-oriented design tools only is not enough. More software can be reused from each application if time is spent during the design phase by identifying and designing components and frameworks. This is the result of abstracting re-usability from applications while building them.

Components (Figure 1-8) are entities that can be used in a number of different programs. Items, such as lists, arrays, and strings, are components of many different programs. The primary goal when designing components is to make them general, so they can be components of as many different applications as possible. Application developers that use components do not need to understand the implementation of those components. They are reusable code in its simplest form.

Components are typically discovered when programmers find themselves repeatedly writing similar pieces of code. Although each piece has been written to accomplish a specific task, the tasks themselves have enough in common that code written to accomplish them appears remarkably alike. When a programmer takes the time to abstract the common elements from the disparate pieces into one, and create a uniform, generally useful interface to it, a component is born. Ultimately, programmers can aim to abstract out common functionality as they design a piece of software before they code similar pieces.

*Frameworks* are skeletal structures of programs that must be fleshed out to build a complete application. The goal when designing frameworks is to make them refinable. The interface to the rest of the application must be as clear and precise as possible. Application developers must be able to quickly understand the structure of a framework, and how to write code that fits within the framework. Frameworks are reusable designs as well as reusable code.

*Applications* are complete programs, similar to a fully-developed simulation, a word processing system, a spread sheet, a calculator, or an employee payroll system. The goal when designing applications is to make them maintainable. This assures that the behavior of the application remains appropriate and consistent during its lifetime. Application developers must frequently make ingenious use of components and frameworks to fit existing systems. Applications must be made compatible with existing software, files, and peripherals so as not to render a smoothly functioning system prematurely obsolete. This requirement makes the design of useful components and frameworks all the more important. If an application is successful, it is maintained and extended in the future. And if an application-specific object has potentially a broader utility, you should consider designing it as a component that can be reused by other applications.

A large by-product of the reuse concept is increased quality. If 90 percent of a new application consists of proven, existing components, only the remaining 10 percent of the code has to be tested from scratch. This, in turn, leads to an increase in maintenance ease. If there are only 10 percent as many defects to begin with, there are a lot fewer bugs to check after the software is in the field. Additionally, the encapsulation and information hiding provided by objects serves to eliminate many kinds of defects and make others easier to find.

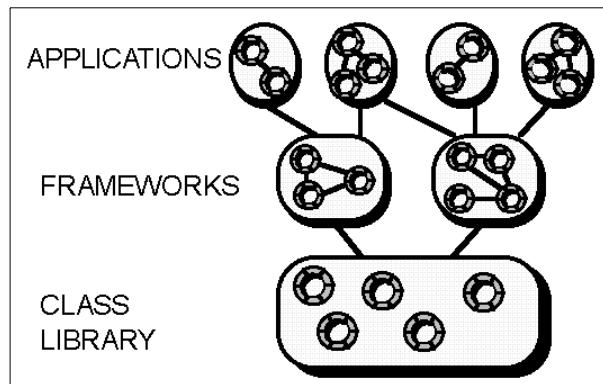


Figure 1-8 Object-oriented development components

In summary, object-oriented technology simulates the real world. Objects are software packages that contain methods/functions (behavior) and variables (state).

Object-oriented technology delivers the following benefits (Figure 1-9):

- ▶ Faster application delivery
- ▶ Higher quality applications
- ▶ Easier maintenance
- ▶ Applications with advanced functions

These benefits are accomplished by implementing the following concepts:

- ▶ Inheritance down the class hierarchy (code reuse)
- ▶ Polymorphism (easier application changes)
- ▶ Encapsulation (easier application changes)
- ▶ Assembly from parts (building quality into the application)

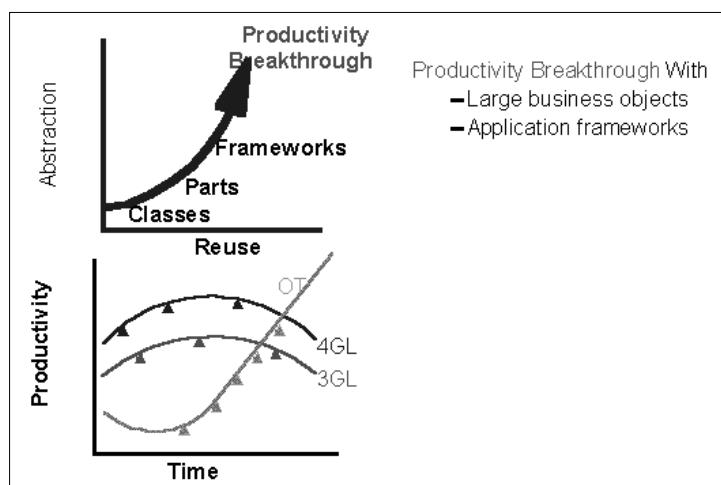


Figure 1-9 Object-oriented technology benefits

## 1.2 Java overview and iSeries implementation

This section discusses the Java platform architecture. First, it looks at how Sun Microsystems, Inc. defines the Java platform and implements it in their Java Development Kit (JDK). Then, it presents the Java implementation on the iSeries server and explains some of the iSeries-specific aspects of this implementation.

## 1.3 Java platform

Java is a full-fledged object-oriented (OO) programming language. The Java language syntax is similar to the syntax of C or C++, while its behavior is more closely related to Smalltalk. Some features of the Java language, such as strongly typed data definitions, no direct memory addressing through pointers, or automatic garbage collection, make it well suited to develop robust enterprise core business applications.

Although Java can be seen as another programming language, it is easy to learn, simple to debug, and has reduced maintenance costs. The main advantage of Java is its cross-platform portability. Java is portable, because of the core Java application programming interfaces (APIs) or Java classes that provide a rich set of platform-neutral APIs. The existence of this set of Java APIs provides the I/T industry with the ability to develop sophisticated, state-of-the-art, client or server Internet-enabled applications that you can deploy and run on any Java-enabled platform. Therefore, Java is not only a new promising programming language. It is a new software platform that you can implement and run on any of the existing hardware or software platforms.

The Java platform can be seen as the combination of three main components:

- ▶ The Java virtual machine (JVM)
- ▶ The Java APIs
- ▶ The Java Utilities

### 1.3.1 Java virtual machine

The Java virtual machine is the centerpiece of the Java platform. It is the “engine” of Java. The JVM is responsible for running Java programs in any given hardware or software environment and is platform-dependent. Every company that wants to implement Java on a given platform must implement the JVM on its hardware or software platform. The JVM generally includes these components:

- ▶ Class loader
- ▶ Bytecode verifier
- ▶ Bytecode interpreter
- ▶ Garbage collector
- ▶ Java Native Interface (JNI)
- ▶ Other miscellaneous components

#### Class loader

The class loader is capable of dynamically locating and loading the various classes that the application uses. This is a powerful feature, because it allows programmers to develop programs that consist of many classes that can be provided by several different vendors. As the acceptance of Java grows in the entire industry, many companies are developing standard, ready-to-use software components or JavaBeans that greatly simplify the job of application developers.

The dynamic nature of the class loader simplifies the application packaging process, because you no longer are required to go through the complex and error-prone process of building the executable program. You can compile each class separately from the other classes and load each class as required by the class loader. However, for performance reasons, developers tend to package related classes together in what is known as *JAR (Java ARchive)* files.

A JAR file is a compressed (zipped) package that includes several classes. The process of packaging an application into a JAR file is simple and much easier than the traditional building steps that were required when using more conventional languages, such as C or C++. Often, all of the classes that make up an application are packaged in a single JAR file. For example, all of the Java Core API classes are shipped within a single file named *classes.zip*. The class loader is capable of finding the required class within a JAR file and expanding (unzipping) it on the fly (instantly).

#### Bytecode verifier

The bytecode verifier performs extensive checks before running a Java program to ensure that the Java bytecodes have not been altered and that they still conform to Java security specifications. This involves such checks as type matching. For example, when an arithmetic operation code is encountered, the bytecode verifier checks that all the operands involved in the operation are of the *integer* type. If the bytecode does not pass the verifier checks, the JVM throws a runtime exception, and the program is terminated. It is especially important to perform extensive checks in an open network environment where someone could run an applet from an unknown source.

#### Bytecode interpreter

The bytecode interpreter is responsible for reading the bytecodes and carrying out the operations that they specify. The bytecodes are interpreted on the fly as the Java application steps from one instruction to the next. As new versions of the JDK are introduced, the overall performance of the bytecode interpreter improves, because of the new algorithms that are being developed.

## **Garbage collector**

The garbage collector provides fully automated memory allocation and de-allocation, which is unlike C++, where the programmer is responsible for allocating memory to store new objects and freeing unused memory when objects are being discarded. The garbage collector solves one of the main problems found in many C++ applications, which is known as memory leaks. This is one of the most difficult bugs to deal with when developing C++ applications, and often C++ applications fail with an “out of memory” error because of poor memory management. In Java, the JVM allocates the memory needed when a new object is created, while a background task, running in a separate thread, continuously scans the memory and de-allocates space that objects (without an active reference in any of the running classes) occupy. The garbage collector is key to both the performance and the reliability of Java programs.

## **Java Native Interface**

You can view the Java Native Interface as “glue” code that allows a Java program to start a method that is written in a language other than Java. Usually, the supported languages are C or C++. This interface allows for interoperation between Java applications and legacy applications. However, by using native methods, you lose portability. By construction, native methods are written to a specific execution environment and are platform-dependent. As soon as a Java application uses code that is written in a different language, the entire application becomes platform-dependent. If portability is important to you, *do not* use JNI.

## **Miscellaneous components**

Some Java implementations may include other components. One of the most commonly found components is a Just-In-Time (JIT) compiler. This is often tightly integrated with the bytecode interpreter. It performs additional tasks such as setting aside in memory the real instructions that correspond to the bytecodes. Any further reference to a bytecode that was executed once results in the execution of the corresponding real machine instruction that already exists. JIT compilers also perform code optimization functions to further improve performance. Functions, such as inlining (which includes a piece of code in another sequence rather than performing a branch or a call to a subroutine and dynamic dead code elimination) are becoming common. In fact, sophisticated compiler optimizing techniques are being implemented in JIT compilers.

Other functions, such as *reflection* or *serialization*, are also found in a JVM. Reflection in Java refers to the ability of a Java class to reflect upon itself (to “look inside itself”). The reflection technique allows a Java program to inspect and manipulate any Java class. This is the technique that the JavaBeans “introspection” mechanism uses to determine the properties, events, and methods that a bean supports. You can use reflection to query and set the values of fields, start methods, or create new objects. Java does not allow methods to pass directly as data values, but the reflection technique makes it possible for methods that pass by name to start indirectly.

*Serialization* is the ability to write the complete state of an object (including any object to which it refers) to an output stream, and then to recreate that object at a later time by reading its serialized state from an input stream. This technique is used as the basis for transferring objects through cut-and-paste and between a client and a server or vice versa for remote method invocation (RMI). It can also be used by JavaBeans to provide pre-initialized serialized objects rather than a simple class file. This technique is also used as an easy way to save users preferences and application states. Serialization includes information about the class version. Obviously, an early version of a class may not be able to de-serialize a serialized instance that was created by a newer version of the same class. See the *serialver* Java utility in 1.3.3, “Java utilities” on page 13.

### 1.3.2 Java APIs

The Java platform provides a set of Java classes or APIs that mimic a complete modern, yet platform-neutral, operating system. The Java platform, which is based on the Sun Microsystems, Inc. JDK, consists of two kinds of APIs:

- ▶ **Core Library APIs** belong to the minimal set of APIs that form the standard Java platform. Core Library APIs are available on the Java platform, regardless of the underlying operating system. They can run on smaller, dedicated embedded systems such as set-top boxes, printers, copiers, and cellular phones. The core library grows with each release of the JDK.
- ▶ **Standard Extension Library APIs** are a set of APIs outside of the Core API for which JavaSoft has defined and published an API standard.

#### Core Library

The classes included in the Core Library are grouped into several packages. Some of the packages in the Core Library are:

- ▶ **java.applet**: Defines the environment in which applets are running. It provides all of the controls that are required for the browser to start the applet.
- ▶ **java.awt**: The Abstract Windowing Toolkit (AWT) provides the basic constructs that are required to build and manage a graphical user interface (GUI).
- ▶ **java.beans**: JavaBeans are reusable software components that conform to the bean specification and are designed to be directly manipulated by visual development tools.
- ▶ **java.io**: Provides all of the constructs that are required to perform standard input and output operations on UNIX-style stream files.
- ▶ **java.lang**: This is the basic Java language package. It includes the definition of all the Java data types and execution behavior, such as multi-threading and exception handling.
- ▶ **java.math**: Enhances the basic language constructs by providing a BigDecimal data type and a BigInteger data type, and associated operations.
- ▶ **java.net**: Provides a Transmission Control Protocol/Internet Protocol (TCP/IP) connectivity environment that includes sockets, uniform resource locator (URL), Hypertext Transfer Protocol (HTTP), and datagram management.
- ▶ **java.rmi**: Remote method invocation allows for interoperation among distributed Java objects.
- ▶ **java.security**: Provides the classes that are required for protecting data exchange on the network by means of private and public key encryption, authentication certificates, and electronic signatures.
- ▶ **java.sql**: This is also known as Java Database Connectivity (JDBC). It is the Java equivalent of Open Database Connectivity (ODBC) and provides all of the constructs that are required to handle relational database accesses.
- ▶ **java.text**: Includes all of the classes that are necessary to develop National Language-enabled applications.
- ▶ **java.util**: Provides basic U.S. English-only date and time functions and a set of utilities, such as string tokenization, hash table, random number generator, and so on.

Some of these packages are also known as the *Enterprise APIs*. Java Enterprise APIs support connectivity to enterprise databases and legacy applications. With these APIs, corporate developers build distributed client and server applets and applications in Java that run on any operating system or hardware platform in the enterprise.

Java Enterprise currently encompasses four areas:

- ▶ **JDBC**: Java Database Connectivity, which is implemented in the `java.sql` package.
- ▶ **RMI**: Remote method invocation, which is implemented in `java.rmi`.
- ▶ **IDL**: Interface Definition Language, which is a CORBA-compliant set of interfaces that provide for seamless integration with CORBA-compliant distributed objects. To provide sufficient time to standardize Java-to-CORBA connectivity through the Object Management Group (OMG), Java IDL will be available on a slightly delayed schedule.
- ▶ **JNDI**: Java Naming and Directory Interface, which provides a unified interface to multiple naming and directory services in the enterprise.

JNDI is part of the Standard Extension library. JDBC, IDL, and RMI are part of the Core library.

In the Java 2 release of the JDK, the Java Foundation Classes (JFC) incorporate several new features that further enhance a developer's ability to deliver scalable, commercial, and mission-critical applications:

- ▶ New high-level GUI components
- ▶ Pluggable look and feel
- ▶ Accessibility support for people with disabilities
- ▶ 2D APIs
- ▶ Drag-and-drop

The JFC is part of the Core library of the Java platform. It extends the original AWT by adding a comprehensive set of GUI class libraries that are portable and compatible with all AWT-based applications.

## Standard Extension library

The Standard Extension library includes all of the Java APIs that have been defined by Sun Microsystems, Inc. and are not part of the Java Core library. These sets of APIs are currently defined:

- ▶ **Java Server APIs**: Are an extensible framework that enables and eases the development of an entire spectrum of Java-powered Internet and intranet servers. The APIs provide uniform and consistent access to the server and administrative system resources that are required for developers to quickly develop their own Java servers.
- ▶ **Java Servlet APIs**: Enable the creation of Java servlets. This API allows developers to incorporate the power of servlets into their existing Web configurations. The Java Servlet Development Kit includes a servlet engine for running and testing servlets, the `java.servlet.*` sources, and all of the API documentation for `java.servlet.*` and `sun.servlet.*`.
- ▶ **Java Commerce APIs**: Bring secure purchasing and financial management to the Web. JavaWallet is the initial component that defines and implements a client-side framework for credit card, debit card, and electronic cash transactions.
- ▶ **JavaHelp APIs**: Are the help system for the Java platform. It is a Java-based, platform-independent help system that enables Java developers to incorporate online help for a variety of needs, which include Java components, applications, applets, desktops, and Hypertext Markup Language (HTML) pages.
- ▶ **Java Media and Communications APIs**: Meet the increasing demand for multimedia in the enterprise by providing a unified, non-proprietary, platform-neutral solution. This set of APIs supports the integration of audio and video clips, animated presentations, 2D fonts, graphics, and images, as well as, 3D models and telephony. By providing standard players and integrating these supporting technologies, the Java Media and Communications APIs enable developers to produce and distribute compelling, media-rich content. The Java

Media and Communications APIs consist of these APIs: Java 2D, Java 3D, Java Media Framework, Java Sound, Java Speech, and Java Telephony.

- ▶ **Java Management APIs:** Provide a rich set of extensible Java objects and methods for building applets that can manage an enterprise network over the Internet. It has been developed in collaboration with SunSoft and a broad range of industry leaders that include: AutoTrol, Bay Networks, BGS, BMC, Central Design Systems, Cisco Systems, Computer Associates, CompuWare, LandMark Technologies, Legato Systems, Novell, OpenVision, Platinum Technologies, Tivoli Systems, and 3Com.
- ▶ **PersonalJava APIs:** Are designed for network-connectable applications on personal consumer devices for home, office, and mobile use. Devices suitable for PersonalJava include: hand-held computers, set-top boxes, game consoles, mobile hand-held devices, and smart phones.
- ▶ **EmbeddedJava APIs:** Are designed for high-volume embedded devices, such as mobile phones, pagers, process control, instrumentation, office peripherals, network routers, and network switches. EmbeddedJava applications run on real-time operating systems and are optimized for the constraints of small-memory footprints and diverse visual displays.

Some of the previously listed APIs may move from the Standard Extension library to the Core library as new releases of the JDK are introduced.

This list gives you an idea of the extensive set of functions that the Java platform now provides and shows some of direction about where Java is going. It demonstrates that Java is not just another simple and easy to use programming language, but a sophisticated programming environment for developing applications now and in the future.

### 1.3.3 Java utilities

The Java utilities are a set of programmer aids that cover the programming side of the application development cycle. These tools are provided:

- ▶ `java`
- ▶ `javac`
- ▶ `jdb`
- ▶ `javah`
- ▶ `javap`
- ▶ `javadoc`
- ▶ `jar`
- ▶ `javakey`
- ▶ `appletviewer`
- ▶ `rmiic`
- ▶ `rmiregistry`
- ▶ `serialver`
- ▶ `native2ascii`

#### The `java` command

The `java` command allows you to run a Java application.

**Note:** Any arguments that appear after the class name on the command line are passed as parameters to the main method of the class. The `java` command expects the binary representation of the class to be in a file called `classname.class`, which is generated by compiling the corresponding source file with the `javac` tool. All Java class files end with the file name extension `.class`, which the compiler automatically adds when the class is compiled. The class must contain a main method defined as shown here:

```
class classname {  
    public static void main(String argv[]){  
        . . .  
    }  
}
```

## The javac command

The **javac** command starts the Java compiler. The Java compiler checks the syntax of a Java source file (.java file). Then, it compiles it and creates a Java bytecode file (.class file) that you run by using the **java** command. Java source code must be contained in files whose file names end with the .java extension. The file name must be constructed from the class name, such as *classname.java* if the class is public or is referenced from another source file. For every class that is defined in each source file and compiled by the **javac** command, the compiler stores the resulting bytecodes in a class file with a name, such as *classname.class*. Unless you specify the -d option, the compiler places each class file in the same directory as the corresponding source file.

## The jdb command

The **jdb** command starts the Java language debugger to help you find and fix bugs in Java programs. The Java debugger is a dbx-like command line debugger for Java classes. It uses the Java Debugger API to provide inspection and debugging of a local or remote Java interpreter.

As with dbx, there are two ways that you can use the **jdb** command for debugging. The most frequently used way is to have the debugger start the Java interpreter with the class that you want to debug. You do this by entering the **jdb** command instead of the **java** command on the command line. The second way to use the debugger is to attach it to a Java interpreter that is already running. For security reasons, the Java interpreter can only be debugged if it is started with the -debug option. When started with the -debug option, the Java interpreter prints out a password that you must specify when starting the debugger with the **jdb** command.

## The javah command

The **javah** command reads a Java class file and creates a C-language header file and stub file to include in a C source file. This utility provides the “glue code” that allows you to call native C or C++ methods from within a Java program. The generated header and source files are used by C programs to reference object instance variables from native source code. The .h file contains a struct definition whose layout parallels the layout of the corresponding class. The fields in the struct correspond to instance variables in the class. The native method interface, JNI, does not require header information or stub files. You can use the **javah** utility with the -jni option to generate native method function prototypes that are needed for JNI-style native methods. The result is placed in the .h file. If you use native methods, your application is not 100 percent pure Java and, therefore, is not directly portable across platforms. Native methods are, by nature, platform or system specific.

## The javap command

The **javap** command starts the Java disassembler, which disassembles a class file. Its output depends on the options that you use. If you do not specify any options, the **javap** command prints out the public fields and methods of the classes that are passed to it. The **javap** command prints its output to stdout. This tool may be useful when the original source code is no longer available and you need to reverse engineer a Java class. However, using this tool may violate the license agreement for the class that you are disassembling.

## **The javadoc command**

The **javadoc** command produces standard documentation for your Java classes. The documentation is in HTML format and can be viewed by any browser. The **javadoc** command generates one .html file for each .java file and each package it encounters. In addition, it produces a class hierarchy file, named tree.html, and an index of the members, called AllNames.html. If you want the **javadoc** command to produce additional information, you must use the special form of comments in your Java source file. The **javadoc** command comments start with `/**` and end with `*/`. All of the text that is included between the opening `/**` tag and the ending `*/` tag is added to the generated documentation.

## **The jar tool**

The **jar** tool combines several Java class files into a single JAR file. You use JAR (.jar) files to minimize Java applet download time and to simplify the Java application installation on distributed clients and servers. The **jar** tool facilitates the packaging of Java applets or applications into a single archive. When the components of an applet or application (class files, images, and sounds) are combined into a single archive, they may be downloaded by a Java agent, such as a browser, in a single HTTP transaction rather than requiring a new connection for each piece. This dramatically improves download time. The **jar** tool also compresses files using a ZIP-like algorithm, which further improves download time.

## **The javakey tool**

The **javakey** tool adds a digital signature to a JAR file. This allows you to improve the security of your Java applets, because the users of your applets know that they are using authenticated Java code that originated from you. The primary use of this utility is to generate digital signatures for archive files.

A signature verifies that a file came from a specified entity, a signer. To generate a signature for a particular file, the signer must first have a public or private key pair associated with it. It must also have one or more certificates authenticating its public key. Users of the authenticated archive file are called *identities*. Identities are real-world entities, such as people, companies, or organizations that have a public key associated with them. An identity may also have one or more certificates authenticating the public key that is associated with it. A certificate is a digitally signed statement from one entity that says that the public key of some other entity has a particular value. Signers are entities that have private keys in addition to corresponding public keys. Private keys differ from public keys in that they can be used for signing. Prior to signing any file, a signer must have a public and private key pair associated with it and at least one certificate that authenticates its public key.

## **The appletviewer tool**

The **appletviewer** tool allows you to run a Java applet without using a browser. If the HTML page you are viewing references several applets, each applet is displayed in a separate window.

## **The rmic command**

The **rmic** command generates a stub class file and a skeleton class file for Java objects that implement the RMI interface. A stub is a proxy for a remote object that is responsible for forwarding method invocations on remote objects to the server where the actual remote object implementation resides. A client reference to a remote object is actually a reference to a local stub. The stub implements only the remote interfaces, not any local interfaces that the remote object also implements. Because the stub implements exactly the same set of remote interfaces as the remote object itself, a client can use the Java language built-in operators for casting and type checking. A skeleton for a remote object is a server-side entity that contains a method that dispatches calls to the actual remote object implementation.

### The rmiregistry command

The `rmiregistry` command starts a remote object registry on a specified port. The remote object registry is a bootstrap naming service that is used by RMI servers on a host to bind remote objects to names. Then, clients can look up remote objects and make RMIs. You would typically use the registry to locate the first remote object that an application needs to start methods. This object provides application-specific support for finding other objects.

### The serialver command

The `serialver` command returns the serial version ID for one or more classes.

### The native2ascii tool

The `native2ascii` tool converts non-Unicode Latin-1 (source code or property) files to Unicode Latin-1. The Java compiler and other Java tools can only process files that contain Latin-1 and Unicode-encoded (udddd notation) characters. The `native2ascii` utility converts files that contain other character encodings into files that contain Latin-1 and Unicode-encoded characters.

For more details on this set of utilities and for the exact command syntax, please refer to Sun Microsystems, Inc. JDK documentation at: <http://java.sun.com/products/jdk/1.2/docs/>

## 1.4 Java on the iSeries server

Starting with V4R2 of OS/400, the iSeries server implements the Java platform. This implementation fully complies with JDK as defined by Sun Microsystems, Inc. Table 1-1 shows the JDK version supported on the iSeries server.

*Table 1-1 JDK versions supported on OS/400*

OS/400 version	JDK version
V4R2	1.1.4, 1.1.6, 1.1.7
V4R3	1.1.6, 1.1.7, 1.1.8
V4R4	1.1.6, 1.1.7, 1.1.8, 1.2.2, 1.3
V4R5	1.1.6, 1.1.7, 1.1.8, 1.2.2, 1.3
V5R1	1.1.8, 1.2.2, 1.3

As in any other Java platform implementation, the iSeries server also provides these components:

- ▶ Java virtual machine
- ▶ Java APIs
- ▶ Java Utilities

The iSeries Java implementation includes these enhancements:

- ▶ The JVM is integrated into the System Licensed Internal Code (SLIC)
- ▶ Static compilation of class files
- ▶ Dynamic class loading
- ▶ Remote Abstract Windowing Toolkit (AWT)
- ▶ Scalable garbage collector
- ▶ DB2 UDB for iSeries JDBC driver
- ▶ Multi-process design point

Figure 1-10 in the following section shows a high-level diagram of the iSeries Java implementation.

### 1.4.1 iSeries Java virtual machine

On the iSeries server, the JVM is implemented within SLIC, below the Technology Independent Machine Interface (TIMI) as shown in Figure 1-10. It is an integral part of OS/400. When you install OS/400 V4R2 or a more recent release on your machine, you have installed a standard JVM on your system.

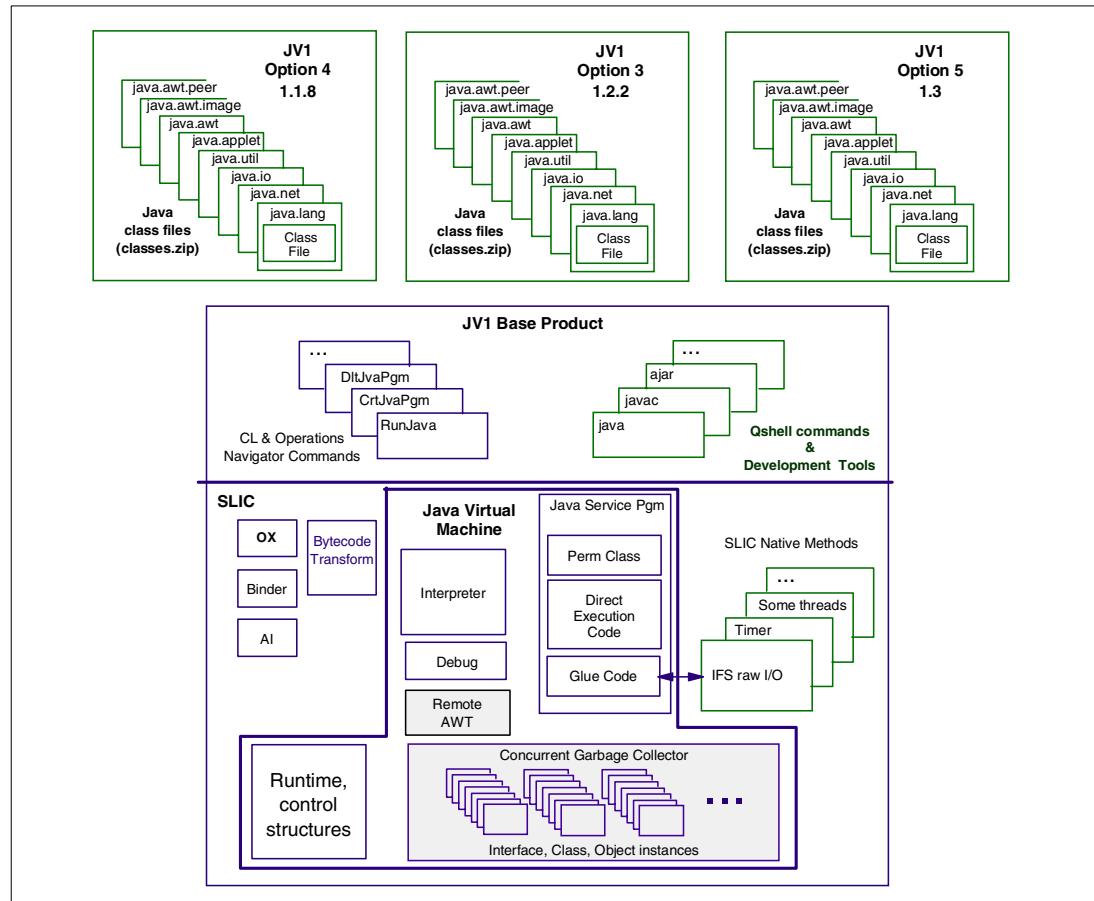


Figure 1-10 iSeries Java virtual machine and IBM Developer Kit for Java

Java brings a number of interesting things to the iSeries server, especially the ability to:

- ▶ Apply OO design principles in a native environment that directly supports those constructs
- ▶ Create Internet applications in a much easier manner than Common Gateway Interface (CGI) programming

Java also fits extremely well with the iSeries server because they both share similar architectural principles, as shown in Figure 1-11 on page 18.

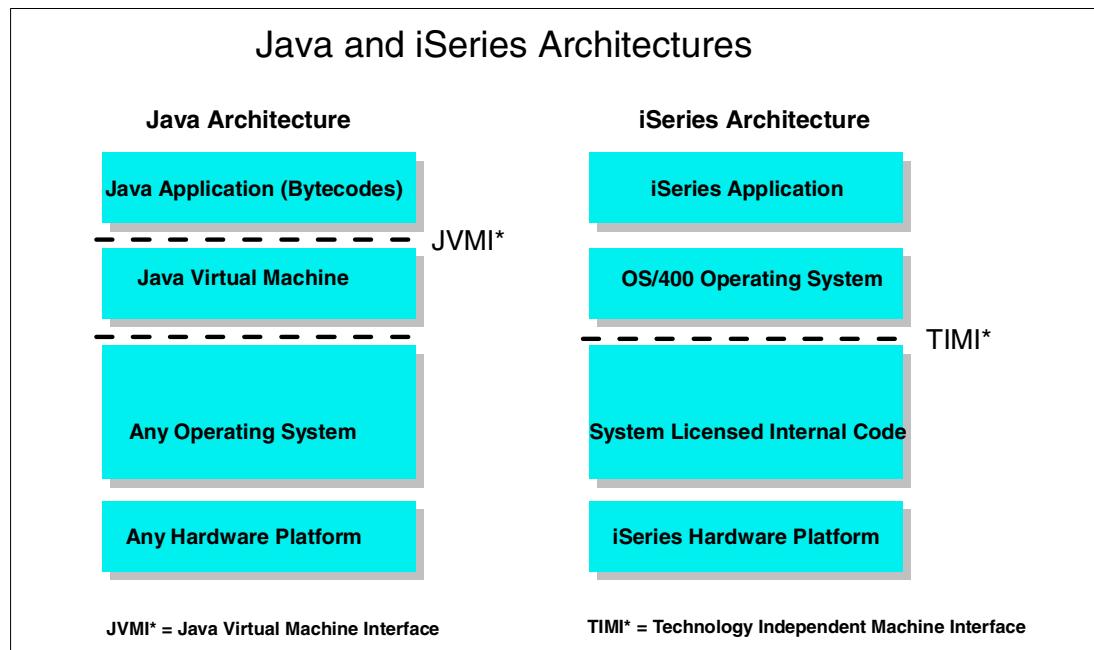


Figure 1-11 Java architecture compared to the iSeries architecture

Figure 1-12 shows a high-level view of how Java is implemented on iSeries by taking advantage of the iSeries architecture.

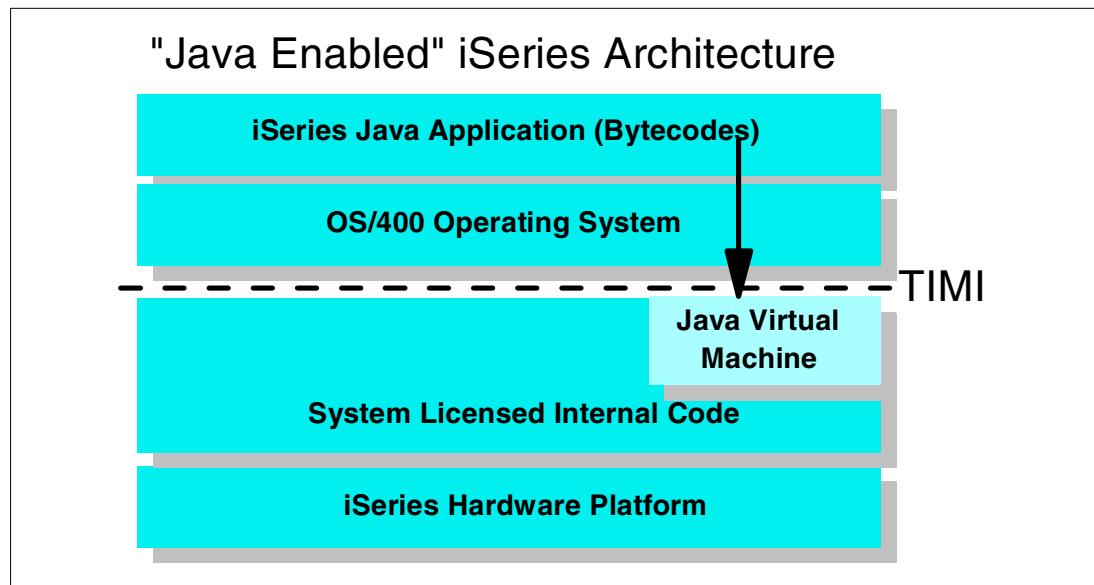


Figure 1-12 High-level schematic of iSeries Java implementation

The OS/400 JVM includes all of these components of a standard JVM as described earlier in this chapter:

- ▶ Class loader
- ▶ Bytecode verifier
- ▶ Bytecode interpreter
- ▶ Garbage collector

The SLIC implementation of the JVM uses the native thread support that was new with the V4R2 release of OS/400. It also supports JNI calls to user-written ILE C or C++ native methods that are packaged in service programs (\*SRVPGM).

**Important:** Although there is no way to prevent a Java program from starting a native ILE C method, which, in turn, calls an RPG or COBOL program, it is not supported in V4R2 or V4R3, because RPG and COBOL are not thread safe. Calling a non-thread safe function from within a threaded environment may cause unpredictable results to occur and should, by all means, be avoided. However, starting with V4R4, you can use RPG and COBOL in native methods and threads, because they are thread safe. For more information about using JNI on the iSeries server, see Chapter 11, “Java Native Interface (JNI)” on page 439.

Not all OS/400 system functions are thread safe. Therefore, using the JNI support to invoke a C function that calls a non-thread safe OS/400 system API may also cause unpredictable results to occur. The *System API Reference V4R4*, SC41-5801, manual has been updated to reflect the thread safe status of every system API. Please refer to this manual before using any system API through the Java JNI support of OS/400.

## 1.4.2 Java APIs and the iSeries server

The Java Core Library APIs, as defined in Sun Microsystems, Inc., are packaged as a separate, no charge, Licensed Program Product (LPP) that you must install on your system to run Java applications on the iSeries server. You need to install the IBM Developer Kit for Java (5769-JV1 in V4R5 and prior releases) or (5722-JV1 in V5R1) LPP product by using the standard OS/400 Install Licensed Program procedures. This is a “skip release” LPP that follows Sun Microsystems, Inc. JDK future versions as closely as possible, independently from OS/400 versions and releases whenever possible.

The *java.io* APIs are linked to the integrated file system support of OS/400 and provide access to any UNIX or PC-style stream file within the integrated file system.

The *java.net* APIs use the standard TCP/IP support that is provided by the TCP/IP Connectivity Utilities for iSeries (5769-TC1) or (5722-TC1 in V5R1) LPP product. You must install this no-charge LPP on your system before using Java on your iSeries server.

The *java.sql* APIs use the standard Structured Query Language (SQL) Call Level Interface (CLI) of OS/400 to access the iSeries database.

The *java.awt* APIs use the Remote AWT support of the iSeries JVM to route any GUI operation to a workstation. This is because there are no GUI-capable devices on the iSeries server. See 1.5, “iSeries-specific implementation” on page 22, for more details on Remote AWT.

## 1.4.3 Java utilities and the iSeries server

Most Java utilities are supported on the iSeries server. They are run from within Qshell Interpreter. The Qshell Interpreter is an option of OS/400 that you must install on your iSeries server to run any Java utility on your system. Refer to 1.5.2, “The Qshell Interpreter” on page 32, for more details on Qshell Interpreter.

As of V4R4, the `qsh` command supports these Java utilities:

- ▶ `java`
- ▶ `javac`
- ▶ `javah`

- ▶ javap
- ▶ javadoc
- ▶ rmic
- ▶ rmiregistry
- ▶ serialver
- ▶ jar
- ▶ ajar
- ▶ javakey
- ▶ appletviewer
- ▶ native2ascii

Apart from the exceptions in the following sections, **qsh** supports the syntax and options of the Java utilities, except the **ajar** tool, as they are described in the standard JDK documentation. The following paragraphs explain how the iSeries implementation is different.

## The **java** command

The **java** command runs Java programs that are associated with a specified Java class. The IBM Developer Kit for Java version of this utility *does not* support these options:

- ▶ **-cs, -checksource:** Both of these options tell Java to check the modification times on the specified class file and its corresponding source file. If the class file cannot be found or if it is out of date, it is automatically recompiled from the source.
- ▶ **-debug:** The iSeries implementation provides the ability to debug Java applications using the standard OS/400 debugging tools. Use the equivalent Run Java (**RUNJAVA**) or **JAVA** OS/400 command to debug a Java program. The IBM Distributed Debugger provides a cooperative debugger that is capable of debugging iSeries server Java applications from a workstation.
- ▶ **-noasyncgc:** The iSeries implementation uses a highly scalable garbage collector instead of the standard JDK garbage collector. You can tune the iSeries garbage collector using the new special options.
- ▶ **-noclassgc:** As previously mentioned, the iSeries garbage collector uses its own options. The standard **java** command options do not apply.
- ▶ **-help:** To obtain help on how to use the **java** command, use the **RUNJAVA** or **JAVA** OS/400 command from any OS/400 command entry line and use the standard OS/400 prompt (F4) and help (F1) support.
- ▶ **-prof:** This sends output profiling information to a specified file or to the **java.prof** file in the current directory. The iSeries server has its own performance analysis tools.
- ▶ **-ss:** Stack size does not apply to the iSeries Java runtime environment, because of the iSeries architecture. There is no such thing as maximum stack size in the iSeries environment.
- ▶ **-oss:** Again, there is no such thing as maximum stack size in the iSeries Java runtime environment.
- ▶ **-t:** Use standard OS/400 debug functions if you need to trace the execution of a Java application. Depending on the optimization level you specified on the Create Java Program (**CRTJVAPGM**) command, or on the **RUNJAVA** or **JAVA** OS/400 command, tracing may not be available.
- ▶ **-verify:** The Java bytecodes are verified and then translated into iSeries RISC PowerPC machine instructions when you run the **CRTJVAPGM** command, or the **RUNJAVA** or **JAVA** OS/400 command. All of the standard **java** utility verify options do not apply to the iSeries Java runtime environment.

- ▶ **-verifyremote**: See the preceding -verify option.
- ▶ **-noverify**: See the preceding -verify option.

The iSeries version of the **java** command supports these specific iSeries options:

- ▶ **-secure**: This option tells the iSeries JVM to check for public write access to the directories that are specified in the CLASSPATH. A directory in the CLASSPATH that has public write authority is a security exposure, because someone may store a class file with the same name as the one that you want to run. The class file that is found first is run. A warning message is sent for each directory in the CLASSPATH that has public write authority. If one or more warning messages are sent, an escape message is sent and the Java program is not run.
- ▶ **-gcfrq**: Specifies the iSeries garbage collector collection frequency. It may or may not be honored depending on the system release and model. Values between 0 and 100 are allowed. A value of 0 means to run garbage collection continuously, a value of 100 means to never run garbage collection, and a value of 50 means to run garbage collection at the default or typical frequency. The default value for this parameter is 50. The more often garbage collection runs, the less likely the garbage collector will grow to the maximum heap size. If a program reaches the maximum heap size, a performance degradation occurs while garbage collection takes place.
- ▶ **-gcpty**: Specifies the iSeries garbage collector collection priority. The lower the number is, the lower the priority is. A low priority garbage collection task is less likely to run because other higher priority tasks are running.

In most cases, -gcpty should be set to the default (equal) value or the higher priority value. Setting -gcpty to the lower priority value can inhibit garbage collection from occurring and result in all Java threads being held while the garbage collector frees storage.

The default parameter value is 20. This indicates that the garbage collection thread has the same priority as default user Java threads. A value of 30 gives garbage collection a higher priority than default user Java threads. Garbage collection is more likely to run. A value of 10 gives garbage collection a lower priority than default user Java threads. Garbage collection is less likely to run.

- ▶ **-opt**: Specifies the optimization level of the iSeries Java program that is created if no Java program is associated with the Java class file. The created Java program remains associated with the class file after the Java program is run.
- ▶ **-verbosegc**: A message is displayed for each garbage collection sweep.

You may prefer to use the RUNJVA or JAVA OS/400 command to benefit from the standard OS/400 environment, such as prompting and online help that you are used to. See 1.5.1, “The OS/400 Java commands” on page 23, for more details on OS/400 Java-related commands.

## **The javac command**

The **javac** command compiles Java programs. It is compatible with the **javac** command that is supplied by Sun Microsystems, Inc. with one exception. The -classpath option does not override the default classpath. Instead, it is appended to the system default classpath. The -classpath option overrides the CLASSPATH environment variable.

## **The javah command**

The **javah** command on iSeries only supports the JNI-type of native method invocations. These include:

- ▶ **-jni**: This option causes the **javah** command to create an output file containing JNI-style native method function prototypes in the current working directory. This option does not have to be specified, but -jni type files are always produced.

- ▶ **-td**: If you specify this option, it is ignored by the iSeries server. The JNI-style C language header file is always created in the current working directory. The header file is created as an iSeries stream file (STMF) in the integrated file system current working directory. This stream file must be copied to a Source Physical File Member in the QSYS.LIB file system before it can be included in a C program on the iSeries server. Use the Copy from Stream File (CPYFRMSTMF) OS/400 command to do so.
- ▶ **-stubs**: If you specify this option, it is ignored by the iSeries implementation.
- ▶ **-trace**: If you specify this option, it is ignored by the iSeries implementation.
- ▶ **-v**: Verbose. This option is not supported.

## The **javap** command

The **javap** command on iSeries ignores these options:

- ▶ **-b**: This option is for backward compatibility with previous releases of the JDK. Because the initial implementation of the JDK on the iSeries server is version 1.1.4 of the JDK, it did not make sense to support this option on the iSeries server.
- ▶ **-p**: On iSeries, -p is not a valid option. You must spell out -private.
- ▶ **-verify**: This option is ignored. The **javap** command does not perform verification on the iSeries server.

## The **ajar** tool

The **ajar** tool is an alternative interface to the **jar** tool that you use to create and manipulate JAR files. You can use the **ajar** tool to manipulate both JAR files and ZIP files. If you need to use a ZIP interface, use the **ajar** tool instead of the **jar** tool.

The **ajar** tool supports these functions:

- ▶ Lists the contents of JAR files
- ▶ Extracts from JAR files
- ▶ Creates new JAR files
- ▶ Supports many of the ZIP formats that the **jar** tool supports
- ▶ Supports adding and deleting files in existing JAR files

## The **appletviewer** tool

You can use the **appletviewer** tool to run applets without a Web browser if you use the Remote AWT support. You need to use Remote AWT because the iSeries server does not have any GUI capable devices.

Applets are intended to be small Java applications that run embedded within a Web browser. It does not make sense to run applets on a server. However, you can run applets on the iSeries server using Remote AWT.

## 1.5 iSeries-specific implementation

This section explains some aspects of the Java implementation that are specific to the iSeries server, such as:

- ▶ OS/400 commands that help you perform Java-related functions in a standard iSeries way with command prompting and online help text
- ▶ Qshell Interpreter environment
- ▶ Remote AWT support

### 1.5.1 The OS/400 Java commands

You can use OS/400 commands to perform certain Java-related functions. Some commands, such as the Run Java (RUNJAVA) command or JAVA command, are OS/400 equivalents to existing Java utilities, such as the `java` command. These commands are specific to the iSeries implementation:

- ▶ Create Java Program (CRTJVAPGM) command
- ▶ Change Java Program (CHGJVAPGM) command
- ▶ Delete Java Program (DLTJVAPGM) command
- ▶ Display Java Program (DSPJVAPGM) command
- ▶ Analyze Java Program (ANZJVAPGM) command

**Note:** The iSeries server supports both the Java Transformer and the Just-In-Time (JIT) compiler.

#### The CRTJVAPGM command

The iSeries implementation of Java provides a unique component called the *Java Transformer*. The Java Transformer preprocesses Java bytecodes that are produced by any Java compiler on any platform and contained in a class file, JAR file, or ZIP file to prepare them to run using the OS/400 JVM. The Java Transformer creates an optimized Java program object that is persistent and is associated with the class file, JAR file, or ZIP file. This program object contains RISC PowerPC 64-bit machine instructions. The optimized program object is not interpreted by the bytecode interpreter at runtime, but directly runs when the class is loaded.

No action is required to start the Java Transformer. It automatically starts the first time that a Java class file is run on the system when you use the `java` command from the Qshell Interpreter or the RUNJAVA or JAVA commands.

It is especially important to use the CRTJVAPGM command on JAR files and ZIP files. Unless the entire JAR file or ZIP file has been optimized using the CRTJVAPGM command, each individual class is optimized at runtime and the resulting program objects are temporary.

By using the CRTJVAPGM command on a JAR file or ZIP file, it causes all of the classes that are contained in that file to be optimized and the resulting optimized Java program object to be persistent. This results in a much better runtime performance.

The CRTJVAPGM command creates an iSeries Java program from a Java class file or one or more Java programs from a JAR file. If the file name ends with `.jar` or `.zip`, then it is a JAR file. The resulting Java program object or objects become part of the class file or JAR file object, but cannot be modified directly. When started by the RUNJAVA command or JAVA command, the Java program is run.

With the CRTJVAPGM command, you specify the name of the Java class file that contains the bytecodes of the Java program that you want to create. You may also specify the name of a JAR file or ZIP file that contains several classes that are packaged together. The CLSF parameter is a required parameter that specifies the class file name or JAR file name from which you create Java programs on iSeries. One or more directory names may qualify the class file name.

For the class file name, specify the name of the class file or a pattern for identifying the name of the class files that are used. Specify a pattern in the last part of the name. An asterisk matches any number of characters, and a question mark matches a single character. Enclose the name in apostrophes if it is qualified or contains a pattern. An example of a qualified class file name is:

```
'/directory1/directory2/myclassname.class'
```

An example of a pattern is:

```
'/directory1/directory2/myclass*.class'
```

For the JAR file name, specify the name of the JAR file or a pattern for identifying the name of the JAR files that are used. A file is a JAR file if it ends with .jar or .zip. Specify a pattern in the last part of the name. An asterisk matches any number of characters, and a question mark matches a single character. Enclose the name in apostrophes if it is qualified or contains a pattern. An example of a qualified class name is:

```
'/directory1/directory2/myappname.jar'
```

An example of a pattern is:

```
'/directory1/directory2/myapp*.zip'
```

You also specify the optimization level of the resulting iSeries Java program. For OPTIMIZE(\*INTERPRET), the resulting Java program interprets the class file bytecodes when it starts. For other optimization levels, the Java program contains machine instruction sequences that run when the Java program starts.

Here are the possible values for the OPTIMIZE parameter:

- ▶ **\*INTERPRET:** The Java programs that you create are not optimized. When you start the program, it interprets the class file bytecode. You can display and change variables while debugging.
- ▶ **10:** The Java program contains a transformed version of the class file bytecodes, but has only minimal additional compiler optimization. You can display and change variables while debugging. This is the default value for the OPTIMIZE parameter.
- ▶ **20:** The Java program contains a compiled version of the class file bytecodes and performs additional compiler optimization. You can display variables, but not change them while debugging.
- ▶ **30:** The Java program contains a compiled version of the class file bytecodes and has more compiler optimization than optimization level 20. You can display, but not change variables while debugging. The values that are presented may not be the current value of the variable.
- ▶ **40:** The Java program contains a compiled version of the class file bytecodes and performs more compiler optimization than optimization level 30. All call and instruction tracing is disabled.

**Note:** If your Java program fails to optimize or throws an exception at optimization level 40, use optimization level 30.

The USRPRF parameter specifies whether the authority checking that is done while the program is running should include only the user who is running the program (\*USER) or both the user who is running the program and the program owner (\*OWNER).

The USEADPAUT parameter specifies whether you can use program adopted authority from previous programs in the call stack as a source of authority when the program is running.

The REPLACE parameter allows you to specify whether an existing iSeries Java program should be replaced.

The ENBPFRCOL parameter allows you to specify whether performance data should be collected. Make sure you choose the proper value if you want to analyze the performance of your Java application. The default value of \*NONE disables performance data collection for that class or set of classes.

The possible values for the ENBPFRCOL parameter are:

- ▶ **\*NONE**: The collection of performance data is not enabled. No performance data is to be collected. This is the default for the ENBPFRCOL parameter.
- ▶ **\*ENTRYEXIT**: Performance data is collected for procedure entry and exit.
- ▶ **\*FULL**: Performance data is collected for procedure entry and exit. Performance data is also collected before and after calls to external procedures.

The SUBTREE parameter specifies whether all subdirectories or no subdirectories are processed when looking for files that match the CLSF keyword.

The LICOPT parameter specifies one or more Licensed Internal Code (LIC) compile-time optimization options. Only advanced programmers, who understand the potential benefits and drawbacks of each selected type of optimization, should use this parameter. An example of a Licensed Internal Code option is:

`'AllowInterJarBinding'`

**Note:** You should contact your service representative for more information on how to use these optimizations.

Three new parameters have been added in V5R1:

- ▶ The CLASSPATH parameter allows the user to provide a specific classpath for locating classes when doing inter-JAR binding.
- ▶ The JDKVER parameter can be used to specify a specific JDK to be used when doing inter-JAR binding.
- ▶ The LICOPTFILE parameter specifies a file that contains Licensed Internal Code options.

Figure 1-13 on page 26 shows the OS/400 prompt for the CRTJVAPGM command.

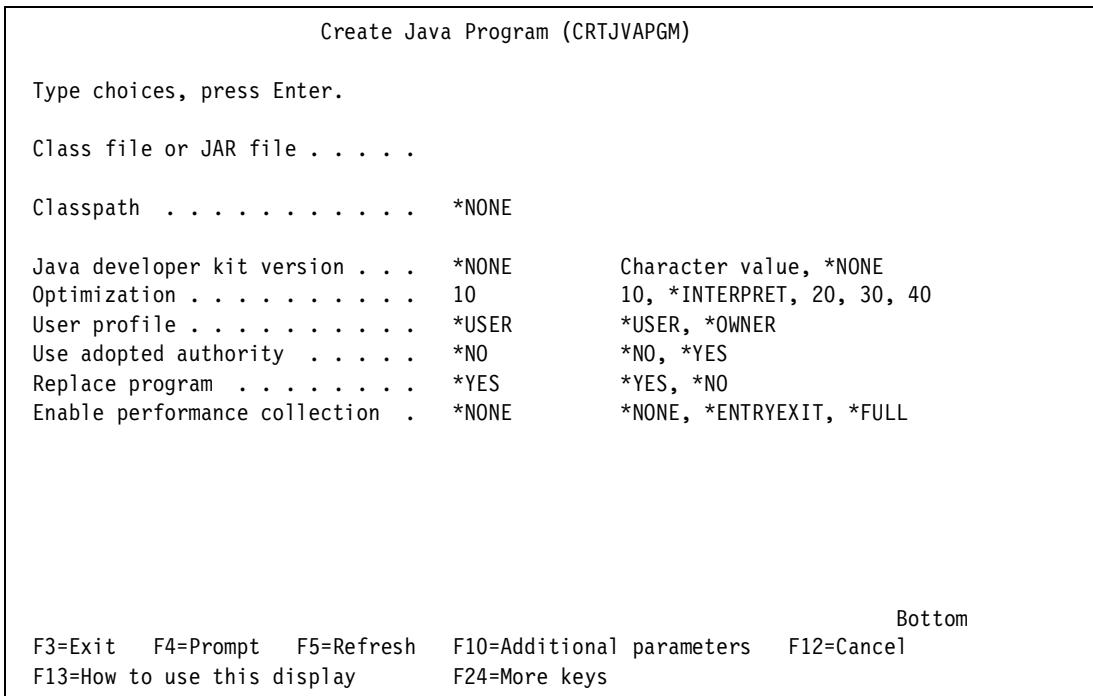


Figure 1-13 Create Java Program (CRTJVAPGM) command

### The CHGJVAPGM command

The Change Java Program (CHGJVAPGM) command changes the attributes of a Java program, which is attached to either a Java class file or a set of Java programs that are attached to a JAR file. A file is a JAR file if the file name ends in .jar or .zip.

The CHGJVAPGM command uses these parameters:

- ▶ CLSF parameter
- ▶ OPTIMIZE parameter
- ▶ ENBPFRCOL parameter
- ▶ LICOPT parameter

You can use these parameters just as you would with the CRTJVAPGM command. See “The CRTJVAPGM command” on page 23 for details about each of these. However, when you use the CHGJVAPGM command, \*SAME (the value does not change) is the default for the OPTIMIZE, ENBPFRCOL, and LICOPT parameters.

In addition, the MERGE parameter specifies whether you merge Java programs, which are attached to a JAR file, into the minimum number of Java programs possible. This parameter is ignored if you are processing a class file. Note these options:

- ▶ **\*RPL:** Specifies that you merge Java programs, which are attached to a JAR file, only if the Java programs need to be recreated and replaced, because other Java program attributes are being changed. If no attributes are changed and no Java programs need to be recreated and changed, merging of Java programs does not occur.
- ▶ **\*YES:** You merge all Java programs, which are attached to a JAR file, into the minimum number of Java programs possible to save space or improve class loader time.

## **DLTJVAPGM command**

The Delete Java Program (DLTJVAPGM) command deletes an iSeries Java program that is associated with a Java class file, JAR file, or ZIP file. If no Java program is associated with the class file that is specified, an informational message (JVAB526) is sent and command processing continues.

## **DSPJVAPGM command**

The Display Java Program (DSPJVAPGM) command displays information about the iSeries Java program that is associated with a Java class file. If no Java program is associated with the class file that is specified, an error message is sent and the command is cancelled. The OUTPUT parameter allows you to specify to where the output should be directed. Specify \* to display the results and \*PRINT to send the results to a spooled file. You can specify the name of a class file, JAR file, or ZIP file.

**Note:** The DSPJVAPGM command previously required no authority to the IFS file to display the Java program. In V5R1, \*R (Read) authority is required.

Figure 1-14 and Figure 1-15 on page 28 show the output of the DSPJVAPGM command.

```
Display Java Program Information
File name . . . . . : < vaapps/workshop/CstmrInqJ.class
Owner . . . . . : ITSCID02

Java program creation information:
File change date/time . . . . . : 04/06/01 16:56:23
Java program creation date/time . . . . . : 04/11/01 10:02:03
Java programs . . . . . : 1
Total classes in source . . . . . : 1
Classes with current Java programs . . . . . : 1
Classes without current Java programs . . . . . : 0
Classes containing errors . . . . . : 0
Optimization . . . . . : 10
Enable performance collection . . . . . : *NONE
Use adopted authority . . . . . : *NO
User profile . . . . . : *USER
                                         More...
Press Enter to continue.

F3=Exit   F12=Cancel
(C) COPYRIGHT IBM CORP. 1980, 2000.
```

Figure 1-14 *Display Java Program (DSPJVAPGM) command (Part 1 of 2)*

```

Display Java Program Information
File name . . . . . : < vaapps/workshop/CstmrInqJ.class
Owner . . . . . : ITSCID02

Licensed Internal Code options . . . . . : AllowInterJarBinding

Java program statistics:
Java program size (K bytes) . . . . . : 28
Release program created for . . . . . : V5R1M0

Bottom
Press Enter to continue.

F3=Exit F12=Cancel

```

*Figure 1-15 Display Java Program (DSPJVAPGM) command (Part 2 of 2)*

### The RUNJVA (or JAVA) command

The Run Java (RUNJVA) command or JAVA command runs the iSeries Java program that is associated with the Java class that is specified. If no \*JVAPGM object is associated with the class file, one is created and associated permanently with the class file. It is used for running the class file, rather than for interpreting the bytecodes.

If you specify the special value \*VERSION on the CLASS parameter instead of a valid Java class name, the build version information for the JDK and the JVM is displayed. No Java program is run.

You can specify these parameters on the RUNJVA (or JAVA) command:

- ▶ **PARM:** Specifies one or more parameter values that are passed to the Java program. You can pass a maximum of 200 parameter values.
- ▶ **CLASSPATH:** Specifies the path that is used to locate classes. Directories are separated by colons. If the special value \*ENVVAR is used, the classpath is determined by the environment variable CLASSPATH. You can set the CLASSPATH environment variable by using the Add Environment Variable (ADDENVVAR) command, be part of an export directive in the system wide /etc/profile file, or specify at the user profile level with an export directive that is contained in the .profile file in the home directory of each user. See Chapter 2, “iSeries Java configuration” on page 37, for more details on setting up the environment.
- ▶ **CHKPATH:** Specifies the level of warnings given for directories in the CLASSPATH that have public write authority. A directory in the CLASSPATH that has public write authority is a security exposure, because it may contain a class file with the same name as the one you want to run. The first class file found is the first class file that is run. The possible values for this parameter are:

- **\*WARN:** A warning message is sent for each directory in the CLASSPATH that has public write authority. This is the default value.
- **\*SECURE:** A warning message is sent for each directory in the CLASSPATH that has public write authority. If one or more warning messages are sent, an escape message is sent, and the Java program does not run.
- **\*IGNORE:** Ignores the fact that directories in the CLASSPATH may have public write authority. No warning messages are sent.
- ▶ **OPTIMIZE:** Specifies the optimization level of the iSeries Java program that is created if no Java program is associated with the Java class file. The created Java program remains associated with the class file after the Java program is run. The possible values for this parameter are identical and have the same meaning as described in “The CRTJVAPGM command” on page 23. You can disable optimization by specifying **OPTIMIZE(\*INTERPRET)** on the RUNJVA (or JAVA) command. This requires that the classes be interpreted regardless of what optimization level you set in the associated Java program object. This is useful if you want to debug a class that was optimized with an optimization level of 30 or 40.
- ▶ **PROP:** Specifies a list of values to assign to Java properties. Up to 100 Java properties can have a value assigned.
- ▶ **GCHINL:** Specifies the initial size (in kilobytes) of the garbage collection heap. This is used to prevent garbage collection from starting on small programs.
- ▶ **GCHMAX:** Specifies the maximum size (in kilobytes) to which the garbage collection heap can grow. This prevents runaway programs that consume all of the available storage. Normally, garbage collection runs as an asynchronous thread in parallel with other threads. If the maximum size is reached, all other threads are stopped while garbage collection takes place.
- ▶ **GCFRQ:** Specifies the iSeries garbage collector collection frequency. It may be honored depending on the system release and model. Values between 0 and 100 are allowed. A value of 0 means to run garbage collection continuously, a value of 100 means to never run garbage collection, and a value of 50 means to run garbage collection at the default or typical frequency. The default value for this parameter is 50. The more often garbage collection runs, the less likely the garbage collector will grow to the maximum heap size. If a program reaches the maximum heap size, a performance degradation occurs while garbage collection takes place.

**Note:** This parameter is no longer supported. It exists solely for compatibility with releases earlier than V4R3 of the iSeries server.

- ▶ **GCPTY:** Specifies the iSeries garbage collector collection priority. The lower the number is, the lower the priority is. A low priority garbage collection task is less likely to run because other higher priority tasks are running.

In most cases, GCPTY should be set to the default (equal) value or the higher priority value. Setting GCPTY to the lower priority value can inhibit garbage collection from occurring and result in all Java threads being held while the garbage collector frees storage.

The default parameter value is 20. This indicates that the garbage collection thread has the same priority as default user Java threads. A value of 30 gives garbage collection a higher priority than default user Java threads. Garbage collection is more likely to run. A value of 10 gives garbage collection a lower priority than default user Java threads. Garbage collection is less likely to run.

**Note:** This parameter is no longer supported. It exists solely for compatibility with releases earlier than V4R3 of the iSeries server.

- ▶ **OPTION:** Specifies special options that you can use when running the Java class. The possible values are:

- **\*NONE**: No special options are used when running the Java class.
- **\*DEBUG**: Allows the iSeries server debugger to be used for this Java program.
- **\*VERBOSE**: A message is displayed each time a class file is loaded.
- **\*VERBOSEGC**: A message is displayed for each garbage collection sweep.
- **\*NOCLASSGC**: Unused classes are not reclaimed when garbage collection runs.

Figure 1-16 shows the RUNJVA command prompt.

```

Run Java Program (JAVA)

Type choices, press Enter.

Class file or JAR file . . . .

Parameters . . . . . *NONE

+ for more values

Classpath . . . . . . . *ENVVAR

Bottom
F3=Exit F4=Prompt F5=Refresh F10=Additional parameters F12=Cancel
F13=How to use this display F24=More keys

```

*Figure 1-16 Run Java Program (RUNJVA) command*

### The ANZJVAPGM command

The Analyze Java Program (ANZJVAPGM) command analyzes a Java program attached to a JAR file, lists its classes, and shows the current status of each class. When specifying **\*FULL** in the **DETAIL** parameter, the bound classes are also shown if they exist. Figure 1-17 shows the ANZJVAPGM command prompt.

Analyze Java Program (ANZJVAPGM)

Type choices, press Enter.

Class file or JAR file . . . . > '/javaapps/workshop/aeusr10/interjarlab.jar'

Additional Parameters

Classpath . . . . . \*PGM

Java developer kit version . . . \*PGM Character value, \*PGM...

Detail . . . . . . . > \*FULL \*NONCURRENT, \*FULL

Output . . . . . . . \* \*, \*PRINT

*Figure 1-17 Analyze Java Program (ANZJVAPGM) command*

Figure 1-18 shows the result of the command. It shows that the JDK classes and the classes from the JAR files are bound into the Java program.

*Figure 1-18 Displaying the Analyze Java Program Information*

## 1.5.2 The Qshell Interpreter

The Qshell Interpreter, **qsh**, is a command interpreter for the iSeries server that is based on POSIX (1003.2) and X/Open (CAE specification for Shell and Utilities) standards. It has many features that make it similar to the Korn shell (**ksh**), and it is upwardly compatible with the Bourne shell (**sh**).

With **qsh**, you can perform these tasks:

- ▶ Run UNIX-like commands from either an interactive session or a script file.
- ▶ Write shell scripts that you can run without modification on other systems.
- ▶ Work with files in any file system that is supported by the integrated file system.
- ▶ Run interactive threaded programs that perform thread safe I/O operations.
- ▶ Write your own utilities to extend the capabilities that are provided by **qsh**.

**qsh** provides these features:

- ▶ Quoting, including the escape character, literal quotes, and grouping quotes
- ▶ Parameters and variables, which includes positional parameters and special parameters
- ▶ Word expansion, which includes tilde (Â) expansion, parameter expansion, command substitution, arithmetic expansion, field splitting, path name expansion, and quote removal
- ▶ Input/output (I/O) redirection
- ▶ Commands, which include pipelines, lists, and compound commands

The current implementation of **qsh** provides built-in utilities for:

- ▶ Defining aliases
- ▶ Working with parameters and variables
- ▶ Running commands
- ▶ Managing jobs
- ▶ Developing Java programs

More utilities will be available in subsequent releases of OS/400.

### Starting the Qshell Interpreter

To start the Qshell Interpreter, use either the Start Qshell (**STRQSH**) command or the **QSH** command. You can specify a Qshell command to run when you start **qsh**. Or, you may start an interactive **qsh** session if you leave the default value \*NONE for the **CMD** parameter on the **STRQSH** command.

If you use the **STRQSH** or **QSH** command in an interactive job, the command starts an interactive shell session. If a shell session is not currently active for your job, then using the **STRQSH** or **QSH** command performs these actions:

- ▶ Starts a new shell session for your job.
- ▶ Displays the shell terminal window on your display.
- ▶ Runs the commands that are contained in the **.profile** file of the **/etc** directory (**/etc/profile**), if such a file exists in the **/etc** directory.
- ▶ Runs the commands that are contained in the **.profile** file of your home directory, if such a file exists in your home directory.

If a shell session is already active for your job, the **STRQSH** or **QSH** command simply reconnects you to the active shell session and displays the shell terminal window on your display.

From the terminal window, you can enter shell commands and view output from the commands that you run. The terminal window has these two parts, which are similar to the OS/400 command entry display:

- ▶ An input line located at the bottom of the display. This allows you to enter shell commands.
- ▶ An output area that contains an echo of the commands you enter on the input line and any output that is generated by the commands that you enter. You can scroll the output area backward and forward.

Figure 1-19 shows the shell terminal window.

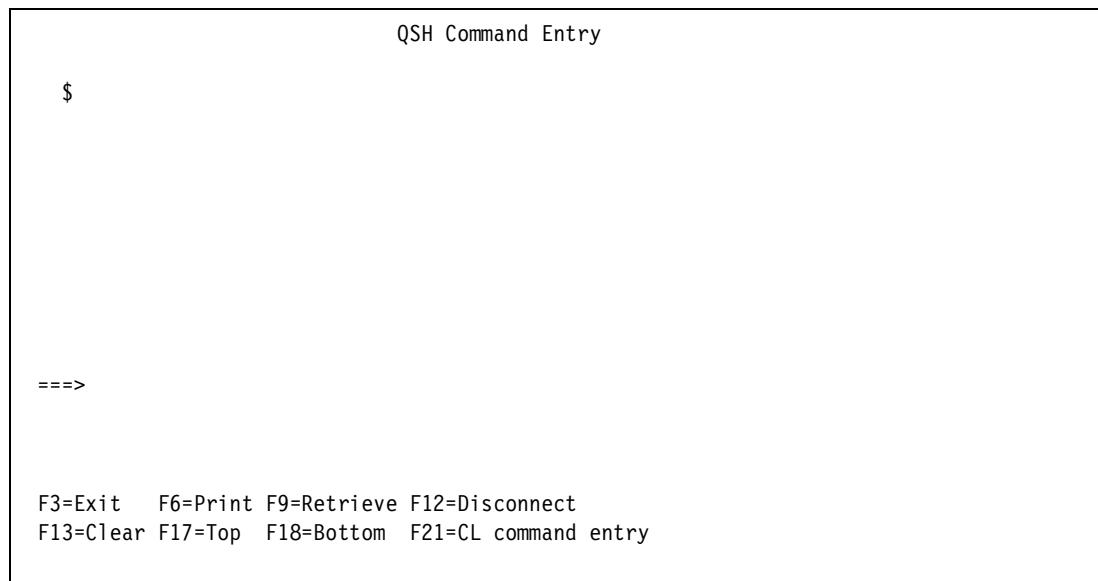


Figure 1-19 QSH Command Entry

These function keys are available on the shell terminal window:

- ▶ **F3=Exit:** Closes the terminal window and ends the Qshell Interpreter session.
- ▶ **F5=Refresh:** Re-displays the contents of the output area.
- ▶ **F6=Print:** Prints the entire contents of the output area to a spooled file.
- ▶ **F7=Up/Page Up:** Displays the previous page of the output area.
- ▶ **F8=Down/Page Down:** Displays the next page of the output area.
- ▶ **F9=Retrieve:** Retrieves the previous command. You can press this key multiple times to retrieve any previous command. You can also select to have a specific command run again by placing the cursor on that command on the output area and pressing the F9=Retrieve key. This copies the selected command from the output area back to the input line where you can modify it as required.
- ▶ **F11=Wrap/Truncate:** Toggles the line wrap or truncate mode for the output area. In line wrap mode, any output longer than the width of the terminal window is wrapped to the next line. In the truncate mode, the portion of the output beyond the width of the terminal window is not shown.
- ▶ **F12=Disconnect:** Closes the terminal window and disconnects your workstation from the Qshell Interpreter session. The `qsh` session does not end and remains active in the background. As soon as you use the STRQSH or QSH command again, you are reconnected to the waiting shell session.

- ▶ **F13=Clear:** Removes all output from previous commands from the output area of the shell terminal display.
- ▶ **F17=Top:** Displays the first page of output data.
- ▶ **F18=Bottom:** Displays the last page of output data.
- ▶ **F19=Left:** Scrolls the display to the left side of the output data.
- ▶ **F20=Right:** Scrolls the display to the right side of the output data.
- ▶ **F21=CL command entry:** Displays a pop-up command entry display where you can enter OS/400 CL commands.
- ▶ **SysReq-2:** Interrupts the currently running shell command.

Please refer to the Qshell Interpreter Reference (<http://as400bks.rochester.ibm.com>) for a complete description of the iSeries Qshell Interpreter features and functions.

### 1.5.3 Remote AWT support

The Remote Abstract Windowing Toolkit (AWT) is an implementation of the Java Abstract Windowing Toolkit. The iSeries server does not have a native GUI device. Remote AWT allows Java applications, which have a graphical user interface, to run on the iSeries server.

The Remote AWT support is a set of Java classes that use the socket support of the JDK. You can use the Remote AWT support to provide a remote display of any Java AWT-based GUI on any Java-compliant platform.

The Java application on the source system (in this case, the iSeries server) uses standard Java AWT APIs to generate a GUI. Every call to any AWT API is passed to the Remote AWT support on the source system. The Remote AWT function uses sockets support to communicate with its equivalent function on the target (remote) system. On the target (remote) system, the Remote AWT support passes all of the AWT requests that it receives from the source system to the standard Java AWT APIs. The standard Java AWT support on the target (remote) system then displays the GUI on the locally attached display device. Keyboard and mouse interactions flow in the opposite direction. They are handled on the target (remote) system by the standard Java AWT APIs and passed to the sockets support. The Remote AWT support sends all the requests to its peer component on the source system by using sockets. On the source system, the Remote AWT support passes all the keyboard and mouse requests back to the standard Java AWT APIs, which, in turn, passes them to the Java application.

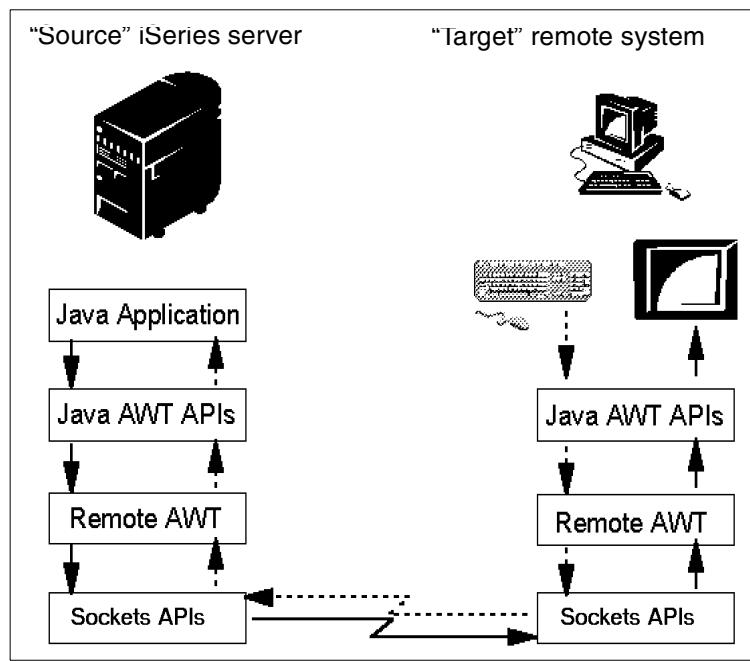


Figure 1-20 Remote AWT (V4R3 and later)

Figure 1-20 shows a Remote AWT implementation. As you can see, the path length involved in the Remote AWT operations is quite long. The sole intent of the Remote AWT support is to provide an implementation that allows any Java application to run on the iSeries server without any modification, even though that application uses Java AWT support while the iSeries does not have any GUI capability.

One of the possible uses of the Remote AWT support is to allow for application installation and configuration, which

usually involves little end-user interaction and is performed on the server.

In V4R3, the iSeries Remote AWT support was changed to use the Java sockets communications interface. In V4R2, it used the RMI interface. The new sockets support provides some performance improvement, but it should still not be used for GUI-intensive applications.

Remote AWT is not intended to be used as a way to support client/server applications involving advanced GUI operations. Such applications must be designed as client/server applications (that is, a client side application that manages the user interface and interacts, using Java RMI APIs with a server-side application that manages database accesses and server side processing). Figure 1-21 shows the architecture of a standard client/server application.

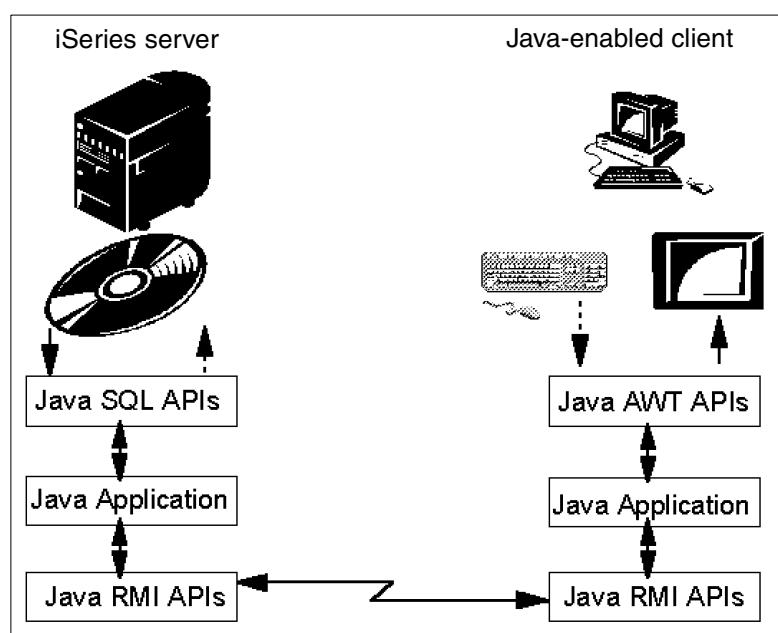


Figure 1-21 iSeries server and Java-enabled client

This is a much “cleaner” design where Java RMI support is only used to communicate between the client-side Java application and the server-side Java application. The overhead incurred is, by far, less important than what is involved in managing a GUI. Also, the Java AWT APIs are used on the client side where the user interaction with the application takes place. In this case, there are no AWT operations performed on the server side.

Conversely, the database operations are performed on the server side. Servers, such as the iSeries server, have an industrial strength database management system (DBMS) that is designed to handle heavy database operations in a secure multi-user environment. Use care when placing enterprise data on client workstations.



# iSeries Java configuration

This chapter provides the information that you need to successfully run Java on the iSeries server. It covers:

- ▶ Checking that the required software is installed
- ▶ Installing the IBM Toolbox for Java on a workstation
- ▶ Setting up the Java runtime environment on the iSeries server
- ▶ Configuring and running Remote Abstract Windowing Toolkit (AWT)

## **2.1 Checking the required software on the iSeries server**

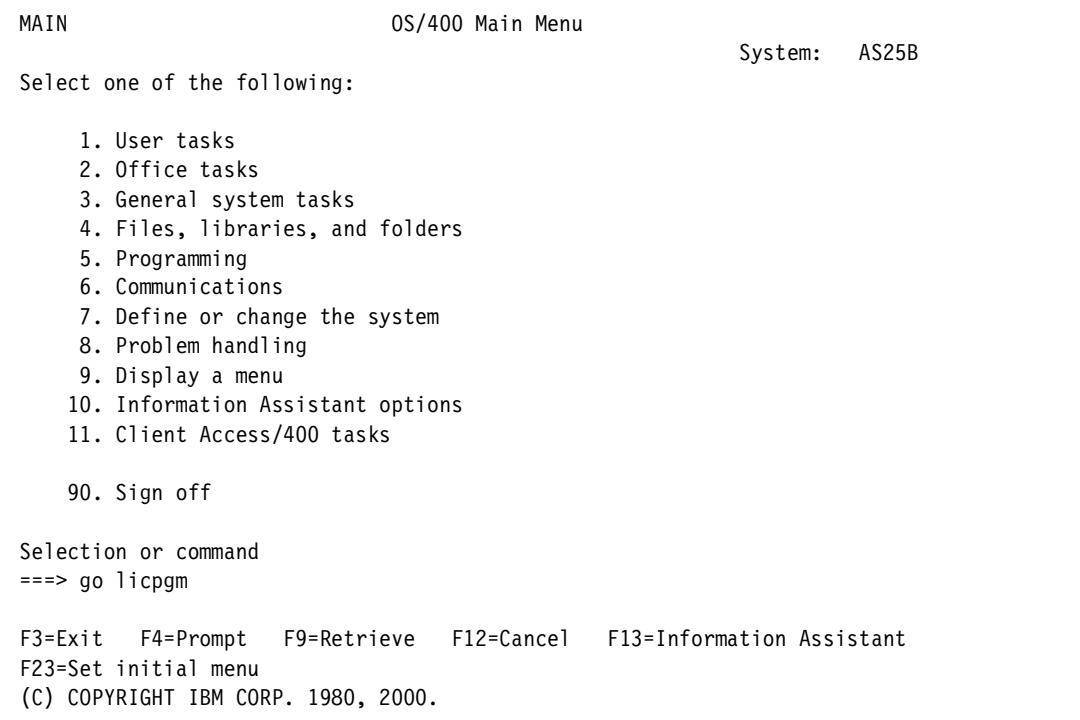
You need to check to see if all of the required software is already installed on the iSeries server. All iSeries servers that are shipped from IBM that have OS/400 V4R2 or higher installed should have all the required support already preloaded. If you are migrating to V4R2 from a previous release of OS/400, you may need to install the required software manually.

To check what software is installed on the iSeries server, perform the following series of steps:

1. Sign on to the system using the QSECOFR user profile as shown in Figure 2-1.

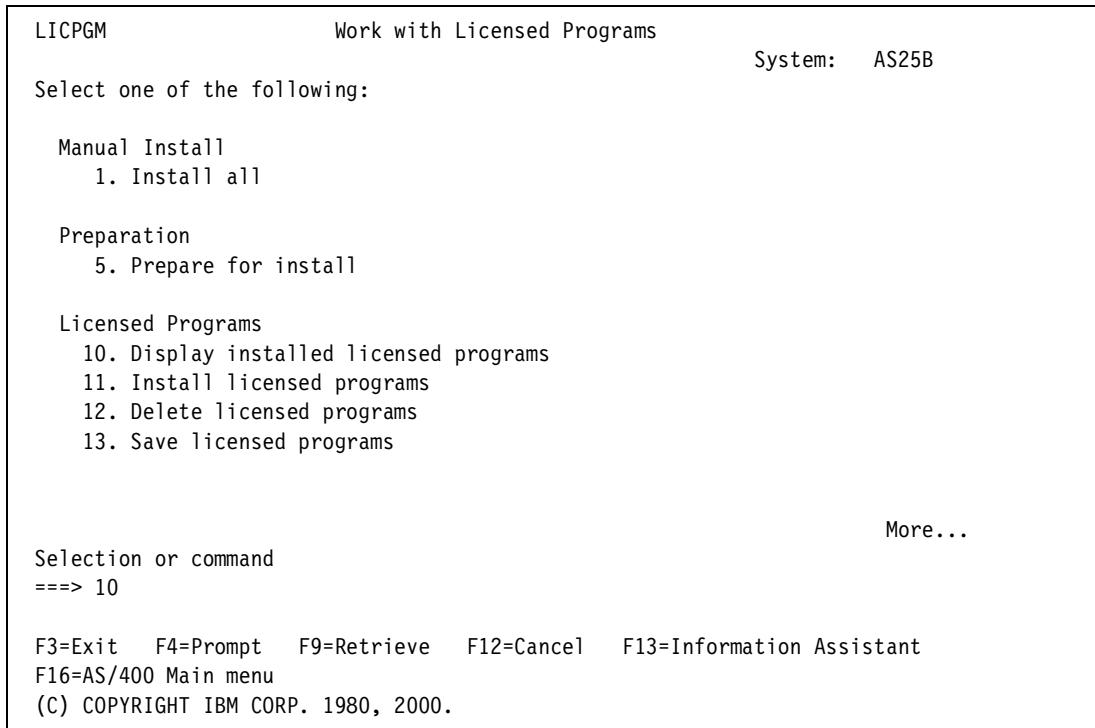
*Figure 2-1 Sign On to the iSeries server using the QSECOFR user profile*

2. Enter the **GO LICPGM** command (as shown in Figure 2-2) on the OS/400 Main Menu command line. The system displays the Work with Licensed Programs menu.



*Figure 2-2 OS/400 Main Menu*

3. Enter option **10** on the Work with Licensed Programs menu command line, as shown in Figure 2-3. All of the LPPs that are currently installed on the system are displayed.



*Figure 2-3 Work with Licensed Program menu*

4. Press the Enter key to show the Display Installed Licensed Programs display. On this display, you can choose the type of information shown in the second column of the display. Here are the options:

- Display the compatibility status of the various LPPs (this is the default setting).
- Press F11 once to display the Installed Release information.
- Press F11 a second time to display the Product Option information.

Since you are looking for OS/400 options, select the Product Option information, as shown in Figure 2-4.

**Note:** The product numbers shown are for V5R1. They will be different for other releases.

Display Installed Licensed Programs		
System: AS25B		
Licensed Program	Product Option	Description
5722SS1		OS/400 - Library QGPL
5722SS1		OS/400 - Library QUSRYS
5722SS1	*BASE	Operating System/400
5722SS1	1	OS/400 - Extended Base Support
5722SS1	2	OS/400 - Online Information
5722SS1	3	OS/400 - Extended Base Directory Support
5722SS1	4	OS/400 - S/36 and S/38 Migration
5722SS1	5	OS/400 - System/36 Environment
5722SS1	6	OS/400 - System/38 Environment
5722SS1	7	OS/400 - Example Tools Library
5722SS1	8	OS/400 - AFP Compatibility Fonts
5722SS1	9	OS/400 - *PRV CL Compiler Support
5722SS1	11	OS/400 - S/36 Migration Assistant
5722SS1	12	OS/400 - Host Servers

More...

Press Enter to continue.

F3=Exit    F11=Display status    F12=Cancel    F19=Display trademarks

Figure 2-4 Display Licensed Programs with Product Options

If you are planning to develop client/server Java applications and run on the iSeries server, you must have OS/400 V4R2 or later. You may require all of the LPPs and operating system options, as shown in Table 2-1.

Table 2-1 Licensed Programs and Product Options

Licensed Program	Product Option	Description
5722-SS1	12	OS/400 - Host Servers
5722-SS1	30	OS/400 - Qshell Interpreter
5722-JC1	*BASE	IBM Toolbox for Java
5722-JV1	*BASE, 3, 4, 5	IBM Developer Kit for Java
5722-TC1	*BASE	TCP/IP Connectivity Utilities
5722-XE1	*BASE	Client Access/400 Express for Windows
5722-XW1	*BASE	Client Access Windows Family Base

Starting with V4R2, Java applications that run on the iSeries server can use the IBM Toolbox for Java (5722-JC1) classes to access iSeries resources, such as data queues, print and spool support, run OS/400 commands, or call programs on iSeries server. For this reason, you must install the OS/400 Host Servers (5722-SS1, option 12), TCP/IP Connectivity Utilities (5722-TC1), and IBM Toolbox for Java (5722-JC1) on the iSeries server.

**IBM Toolbox for Java:** The Licensed Product Product for the IBM Toolbox for Java has changed in V5R1 to 5722-JC1. It is also known as Modification 4 of the IBM Toolbox for Java. Modification 4 can only be installed on iSeries servers at V4R4 or later. From a client, the IBM Toolbox for Java connects back to V4R3 and later of OS/400.

Prior to V4R4, the IBM Toolbox for Java is licensed program product 5763-JC1. It can be used in a client/server environment with systems that run OS/400 V3R2, V3R7, V4R1, V4R2, V4R3, or V4R4.

The IBM Toolbox for Java provides a set of Java classes that simplify the access of iSeries resources from a Java application. You can use these classes in a client Java application or in a server Java application.

In earlier releases, such as V3R2, not all IBM Toolbox for Java features may be available, because certain host server functions are not available. For example, on OS/400 V3R2, DDM is not supported over a TCP/IP connection. You can enable this feature on V3R7 and V4R1 by applying the appropriate PTFs. These Java classes use TCP/IP sockets to connect to the host servers that run on the iSeries server. This is why you must install the OS/400 Host Servers and TCP/IP Connectivity Utilities.

The IBM Developer Kit for Java (5722-JV1) provides the Java classes or application programming interfaces (APIs) that are defined by Sun Microsystems, Inc. The current implementation supports multiple JDKs. You can install more than one JDK at once. You can also run multiple JVMs concurrently, each with a different version of the JDK. In V5R1, you can have three JDK options:

- Install option 3 for JDK 1.2.2
- Install option 4 for JDK 1.1.8
- Install option 5 for JDK 1.3

The Qshell Interpreter (5722-SS1, option 30) provides a character-based command level environment that allows you to use standard Java commands, such as `java`, `javac`, `javadoc`, and so on, from any iSeries workstation. This support allows you to perform Java-related functions, such as compiling Java source code into bytecodes or running a Java program in the same way as you do when performing these tasks on a PC workstation from a DOS session.

An extension to the Qshell Interpreter is also available. Qshell Utilities for AS/400 (5799-XEH) is available as a PRPQ. You must install the Qshell Interpreter before installing the Qshell Utilities. Starting with V4R4, you do not need to install the Qshell Utilities because they are included with the Qshell Interpreter.

5. Use the Page Down key (or the Scroll Up key) to page through the Display Installed Licensed Programs display and look for the required OS/400 options and LPPs. Figure 2-5 shows OS/400 - Qshell Interpreter (5722-SS1, option 30).

Display Installed Licensed Programs		
System: AS25B		
Licensed Program	Product Option	Description
5722SS1	13	OS/400 - System Openness Includes
5722SS1	14	OS/400 - GDDM
5722SS1	16	OS/400 - Ultimedia System Facilities
5722SS1	18	OS/400 - Media and Storage Extensions
5722SS1	21	OS/400 - Extended NLS Support
5722SS1	22	OS/400 - ObjectConnect
5722SS1	23	OS/400 - OptiConnect
5722SS1	25	OS/400 - NetWare Enhanced Integration
5722SS1	26	OS/400 - DB2 Symmetric Multiprocessing
5722SS1	27	OS/400 - DB2 Multisystem
<b>5722SS1</b>	<b>30</b>	<b>OS/400 - Qshell Interpreter</b>
5722SS1	31	OS/400 - Domain Name System
5722SS1	32	OS/400 - Directory Services
5722SS1	33	OS/400 - Portable App Solutions Environment

More...

Press Enter to continue.

F3=Exit F11=Display status F12=Cancel F19=Display trademarks

Figure 2-5 Display Installed Licensed Programs (5722-SS1 option 30)

Browse through the various pages of the displayed LPPs and look for all the programs that are previously listed. Make sure that **IBM Toolbox for Java (5722-JC1)** and **IBM Developer Kit for Java (5722-JV1)** are installed as shown in Figure 2-6.

Display Installed Licensed Programs		
System: AS25B		
Licensed Program	Product Option	Description
5722DG1	*BASE	IBM HTTP Server
5722DG1	1	Triggered Cache Manager
5799GDJ	*BASE	WRKASP Utility Base Product
<b>5722JC1</b>	<b>*BASE</b>	<b>Toolbox for Java</b>
5722JS1	*BASE	Job Scheduler
<b>5722JV1</b>	<b>*BASE</b>	<b>Developer Kit for Java</b>
5722JV1	3	Java Developer Kit 1.2
5722JV1	4	Java Developer Kit 1.1.8
5722JV1	5	Java Developer Kit 1.3
5769LNP	*BASE	Lotus Enterprise Integrator
5769LNT	*BASE	Lotus Domino For AS/400
5769LNT	1	AS/400 Integration
5769LNT	3	C API
5769LNT	4	C++ API

More...

Press Enter to continue.

F3=Exit F11=Display status F12=Cancel F19=Display trademarks

Figure 2-6 Display Installed Licensed Programs (5722-JC1) and (5722-JV1)

Continue paging down until you find the **TCP/IP Connectivity Utilities (5722-TC1)** LPP as shown in Figure 2-7.

Display Installed Licensed Programs		
System: AS25B		
Licensed Program	Product Option	Description
5769RD1	5	OnDemand Server feature
5798RZG	*BASE	AS/400 Service Director
5722SM1	*BASE	System Manager for AS/400
5722ST1	*BASE	DB2 Query Mgr and SQL DevKit
5798TBY	*BASE	IBM Facsimile Support for AS/400
<b>5722TC1</b>	<b>*BASE</b>	<b>TCP/IP Connectivity Utilities</b>
5722WDS	*BASE	WebSphere Development ToolSet
5722WDS	21	Tools - Application Development
5722WDS	31	Compiler - ILE RPG IV
5722WDS	32	Compiler - System/36 Compatible RPG II
5722WDS	33	Compiler - System/38 Compatible RPG III
5722WDS	34	Compiler - RPG/400
5722WDS	35	Compiler - ILE RPG IV *PRV
5722WDS	41	Compiler - ILE COBOL

More...

Press Enter to continue.

F3=Exit F11=Display status F12=Cancel F19=Display trademarks

Figure 2-7 Display Installed Licensed Programs (5722-TC1)

If you plan on using PCs that run Microsoft Windows as Java clients or as Java Remote AWT devices for the iSeries applications, use Client Access Express. This provides easy-to-use iSeries connectivity features, such as accessing the integrated file system to store Java source code and Java bytecodes from the PC. In this case, look for the **Client Access/400 Express for Windows (5722-XE1)** LPP, as shown in Figure 2-8.

Display Installed Licensed Programs		
System: AS25B		
Licensed Program	Product Option	Description
5722WDS	*COMPATIBLE	Compiler - ILE C *PRV
5722WDS	*COMPATIBLE	Compiler - ILE C++ *PRV
5722WDS	*COMPATIBLE	IBM Open Class - source and samples
5722WDS	*COMPATIBLE	Workstation Tools - Base
5722WDS	*COMPATIBLE	Workstation Tools - WebFacing, CODE
5722WDS	*COMPATIBLE	Workstation Tools - VisualAge RPG
5722WDS	*COMPATIBLE	Workstation Tools - WebSphere Studio
5722WSV	*COMPATIBLE	Integration for Windows Server
5733WS4	*INSTALLED	WebSphere App Server, single server
5733WS4	*INSTALLED	WebSphere App Server, Server
<b>5722XE1</b>	<b>*COMPATIBLE</b>	<b>Client Access/400 Express for Windows</b>
<b>5722XW1</b>	<b>*COMPATIBLE</b>	<b>Client Access Family for Windows</b>
5722XW1	*COMPATIBLE	Client Access Enablement Support
0TOOL00	*INSTALLED	STT Tools (*BASE)

More...

Press Enter to continue.

F3=Exit F11=Display status F12=Cancel F19=Display trademarks

Figure 2-8 Display Installed Licensed Programs (5722-XE1) and (5722-XW1)

If the required OS/400 options or LPPs are not installed on the system, you need to go through the steps to manually install any missing OS/400 options or LPPs on your system. For information about how to install the licensed programs or operating system, refer to the software installation guide.

### 2.1.1 Setting up the Java version to use

Java system properties determine the environment in which the Java programs run. They are similar to system values or environment variables in OS/400. A number of properties are set when the Java virtual machine starts. The system properties are set to system default values. The iSeries supports the use of multiple JDKs simultaneously, but only through multiple Java virtual machines. A single Java virtual machine runs one specified JDK.

1. The `java.version` property determines which JDK to run. To determine the default JDK for your system, open a Qshell session and use the `java -version` command.

```
java -version  
java version "1.2.2"
```

Figure 2-9 Determining the JDK version

2. As shown in Figure 2-10, if you want to change the default JDK, use the `SystemsDefault.properties` file. This file can be used to override default Java settings. Add an entry for the `java.version` property. The value you set will override the default setting.

```
java.version=1.1.8
```

Figure 2-10 Overriding the default JDK version setting in the `SystemsDefault.properties` file

3. Place the `SystemsDefault.properties` file in the `/qibm/userdata/java400` directory of the iSeries integrated file system. However, when you change the `java.version` property it has no effect once a JVM is up and running.
4. You can also change the JDK to use at run time using a Java property. As shown in Figure 2-11, open a Qshell session and use the property setting to set the desired JDK version.

```
java -Djava.version=1.3 HelloWorld
```

Figure 2-11 Changing the JDK version at run time

## 2.2 Setting up the environment on the iSeries server

On the iSeries server, there is no requirement to set up any special environment to use the Java support that the JDK provides. There is not an `autoexec.bat` file with a `PATH` environment variable. Similar to any other standard JDKs, the IBM Developer Kit for Java does not require that you have to change the `CLASSPATH` environment variable. The IBM Developer Kit for Java can find the core Java classes as long as they are stored in the `java.zip` and `sun.zip` files that are located in the `/qibm/proddata/java400/jdk11x/lib` subdirectory (x is 6,7 or 8). Starting with V4R4, the `classes.zip` file replaces the `java.zip` and `sun.zip` files. The `jdk 1.2` and `1.3` classes are found in the `rt.jar` file.

The iSeries JVM automatically searches all the classes it needs to load in the `LIB` subdirectory that is located in the `jdk` directory on the iSeries integrated file system.

Therefore, if you do not modify anything in the jdk directory or its associated subdirectories (as shipped with the 5722-JV1 LPP), the Java executables find the classes.

You need to set up the CLASSPATH environment variable if you want to load additional class libraries, such as the ones that you develop yourself or the IBM Toolbox for Java classes.

To use the IBM Toolbox for Java classes, you must tell the JVM where to look for these classes. You can do this in several different ways. The best way to set up the iSeries environment depends on whether you want to run **java** from the Qshell Interpreter, by using Command Language (CL) commands (for example, RUNJVA), or both. This section provides details on how to do this.

## Creating a symbolic link

In these examples, we use a symbolic link. This link is used to simplify setting up the iSeries environment. We do this by entering the following command on an iSeries command entry line:

```
ADDLNK OBJ('/QIBM/ProdData/OS400/jt400') NEWLNK(AS400JT)
```

An example of adding the symbolic link AS400JT is shown in Figure 2-12.

*Figure 2-12 Adding a symbolic link*

Now, rather than entering /QIBM/ProdData/OS400/jt400 each time, you can use AS400JT instead.

### 2.2.1 Setting up the Java environment for CL commands

If you want to use Java CL commands, you can:

- ▶ Specify the CLASSPATH each time you run a CL command.
  - ▶ Use the CLASSPATH environment variable to set the CLASSPATH variable.

## Setting CLASSPATH on every Java CL command

You can specify the proper path on the CLASSPATH parameter on the CRTJVAPGM command and RUNJVA (or the JAVA) command as shown in Figure 2-13.

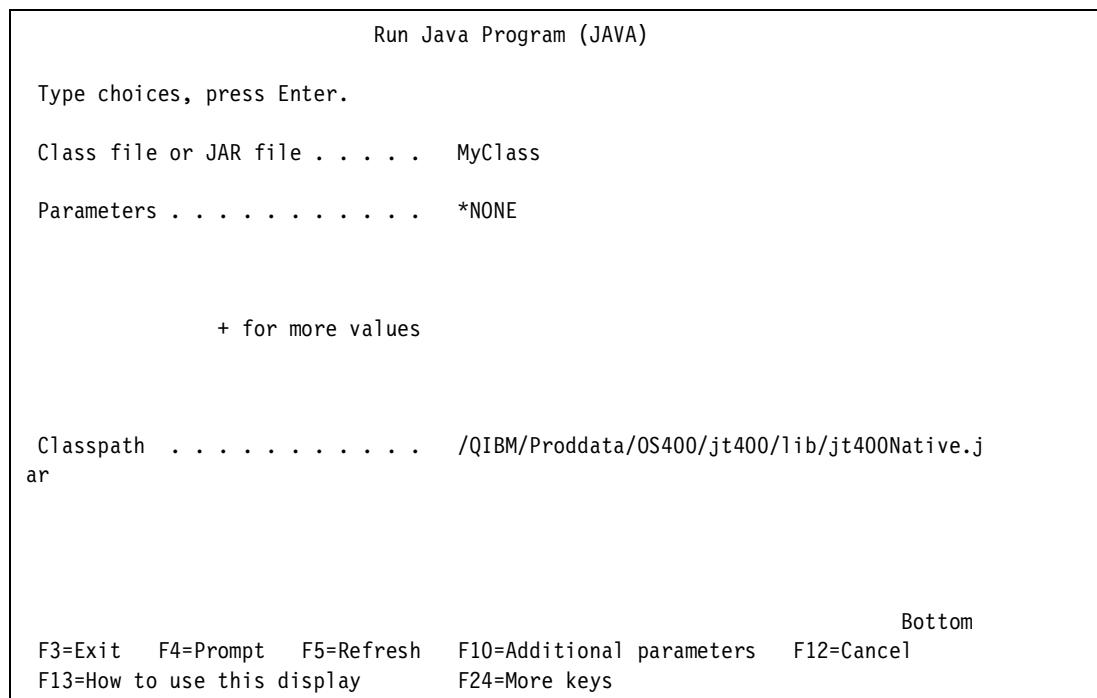


Figure 2-13 Run Java Program display

Since you have to do this every time you compile or run a Java program, it can be error prone and soon becomes quite tedious.

## Setting the CLASSPATH environment variable

The CLASSPATH environment variable is available, and you can use it to set the CLASSPATH. Two CL commands are available:

- ▶ **Add Environment Variable (ADDENVVAR) command:** To add the environment variable for the session.
- ▶ **Work with Environment Variable (WRKENVVAR) command:** To work with an existing environment variable.

There are two levels of this environment variable:

- ▶ \*JOB is a job-level effect.
- ▶ \*SYS is a system-level effect.

If you prefer using the \*JOB option, you can set your iSeries user profile to run a CL program that sets the CLASSPATH environment variable each time that you sign on. For example, this command sets the CLASSPATH environment variable to allow the IBM Toolbox for Java classes to be found:

```
ADDENVVAR ENVVAR(CLASSPATH) VALUE('AS400JT/lib/jt400Native.jar')
```

Now, when a Java CL command is used (for example, the RUNJVA command), the classpath parameter can be set to \*ENVVAR. This is the default setting. It uses the value that you set previously with the ADDENVVAR command. This technique also works when using the Qshell Interpreter. We use the symbolic link, AS400JT, rather than the full path name.

**Note:** This directive is a succession of directory names separated by a colon (:).

You can use several different techniques to set the CLASSPATH for CL commands. The CLASSPATH that you use depends on which technique applies to the user.

The order of precedence for setting the CLASSPATH for CL commands is:

1. CLASSPATH parameter on the CL command
2. CLASSPATH environment variable as set by ADDENVVAR

## 2.2.2 Setting up the environment for the Qshell Interpreter

If you want to use the Qshell environment, you can:

- ▶ Use the CLASSPATH environment variable to set the CLASSPATH variable.  
This method uses the same technique that was described previously for CL commands.
- ▶ Use the export directive to dynamically set up a CLASSPATH.
- ▶ Use the -classpath parameter on a Java command to set up a CLASSPATH dynamically.
- ▶ Set up profile files for individual users.
- ▶ Set up a profile for the entire system.

You can use several different techniques to set the CLASSPATH for an iSeries server. The CLASSPATH that you use depends on which technique applies to the user. The order of precedence for setting the CLASSPATH for the Qshell environment is:

1. The -classpath parameter on a Java command
2. The export directive to set up a CLASSPATH dynamically
3. Profile files for individual users
4. Profile file for the entire system
5. CLASSPATH environment variable as set by ADDENVVAR

### Setting CLASSPATH in Qshell Interpreter

Enter this command to set the CLASSPATH variable after starting the Qshell Interpreter with the Start Qshell (STRQSH) command:

```
export -s CLASSPATH=.:AS400JT/lib/jt400Native.jar
```

The export directive uses the symbolic link called AS400JT, rather than the full path name.

**Note:** This directive is a succession of directory names that are separated by a colon (:). The Qshell Interpreter searches the directories in the order that is specified, from left to right, until it finds the classes to load. The current working directory is specified by a period (.) or a null directory before the first colon. This is set dynamically and is, in effect, for only the current session of Qshell.

### Setting the CLASSPATH using the -classpath parameter

This option is similar to using the export directive, but sets the CLASSPATH variable by using a parameter on a Java command. It is only in effect for the current Java command, for example:

```
java -classpath .:AS400JT/lib/jt400Native.jar MyJavaPgm
```

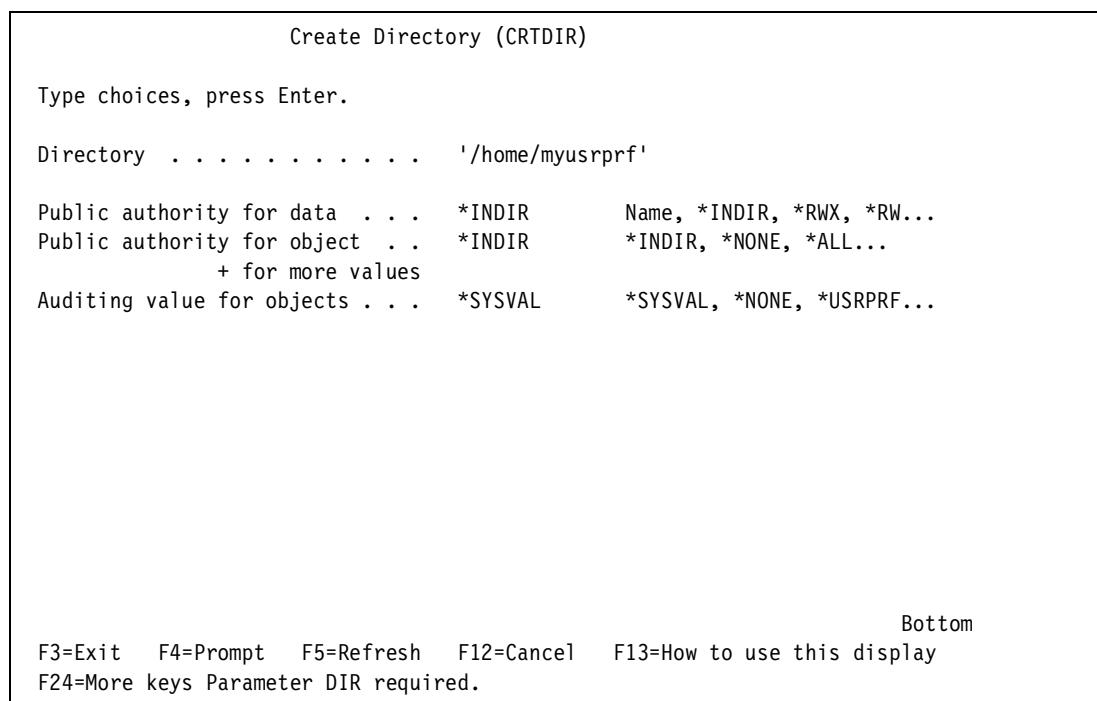
## Setting up Qshell Interpreter for individual users

When using the Qshell Interpreter, you can control the CLASSPATH information by individual user or system wide. This section shows you how to change the CLASSPATH information for a limited number of user profiles. To do so, you must create a .profile file in the *home* directory of every user who wants to access the IBM Toolbox for Java classes. First, you need to create the user home directory. You can do this by entering this command on the iSeries command entry line:

```
CRTDIR DIR('/home/myusrprf')
```

In this command, *myusrprf* is the user profile for the user.

You may also use the PC-like alias of the `md` command or the `mkdir` command. The Create Directory screen appears as shown in Figure 2-14.



*Figure 2-14 Creating an integrated file system directory on the iSeries server*

This creates a subdirectory, named *myusrprf*, in the *home* directory. You need to edit the *.profile* file for that user profile. You can do this by entering this command on the iSeries command entry line:

```
EDTF STMF('/home/myusrprf/.profile')
```

The EDTF command invokes a stream file editor, which is similar to Source Entry Utility (SEU). Add this line in the .profile file for this user:

```
export -s CLASSPATH=.:$HOME/AS400JT/lib/jt400.zip
```

The EXPORT directive uses the symbolic link, called AS400JT, rather than the full path name. This makes it much easier when you want to create .profile files for many users on the system.

**Note:** This directive is a succession of directory names that are separated by a colon (:). The Qshell Interpreter searches the directories in the order that is specified, from left to right, until it finds the classes to load. The current working directory is specified by a period (.) or a null directory before the first colon. The \$HOME parameter represents the user home directory, as created before. This statement allows you to find the IBM Toolbox for Java classes and your own Java classes, as long as your classes are stored in your home directory. Figure 2-15 shows the EDTF window.

```
Edit File: /home/itscid02/.profile
Record :      1   of      1 by    8          Column :      1      59 by  74
Control :

CMD ....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+
*****Beginning of data*****
export -s CLASSPATH=.:${HOME}/as400jt/lib/jt400Native.jar
*****End of Data*****


F2=Save   F3=Save/Exit   F12=Exit   F15=Services   F16=Repeat find
F17=Repeat change   F19=Left   F20=Right
(C) COPYRIGHT IBM CORP. 1980, 2000.
```

Figure 2-15 Using EDTF to create the .profile file

This setup allows you to create an identical .profile file in every user home directory, instead of having a specific .profile file for every user.

Since you use the \$HOME parameter to control the path to the IBM Toolbox classes, you must add a symbolic link in the user directory. To do this, sign on as the user and run the ADDLNK command. Then, create a link, named AS400JT, that points to the IBM Toolbox for Java classes.

### Setting up Qshell Interpreter for the entire iSeries server

You can set up the CLASSPATH environment to use a system-wide profile file that is located in the **etc** directory. Instead of creating one /home/myusrprf/.profile file for every single user that requires access to the IBM Toolbox for Java classes or to your own developed Java classes, you can define a single /etc/profile file that applies to the entire system. You create this file with the same EDTF editor that you used when creating the individual files. Enter this line:

```
export -s CLASSPATH=.:${AS400JT}/lib/jt400Native.jar
```

Similar to the individual user profile setup, we use a symbolic link, named AS400JT. This simplifies the creation of the EXPORT directive and makes it possible to access the IBM Toolbox for Java classes and your own classes, through a single export statement. In this case, you do not use the user home directory to control the path to the IBM Toolbox classes, so you use a system wide symbolic link. In this case, it is named AS400JT.

If you need to create the symbolic link, you create it exactly as explained before by using the ADDLNK command.

### 2.2.3 iSeries CLASSPATH recommendation

You can automate the process of setting the CLASSPATH environment variable by creating a CL program similar to the example in Figure 2-16 and setting it as the Initial Program to Call (INLPGM) in your user profile.

```
PGM  
ADDENVVAR ENVVAR(CLASSPATH)+  
    VALUE('..:/as400jt/lib/jt400Native.jar')  
ENDPGM
```

Figure 2-16 CL source for the initial program

You can add any other files you want in the CLASSPATH to this program. We recommend using this technique to set the CLASSPATH. It works for both CL commands and the Qshell environment. By making this program, the initial program will run each time you sign on. This ensures that the CLASSPATH is set properly each time.

### 2.2.4 Testing the Java environment on the iSeries server

Now, we create a simple “Hello World” Java application that runs on the iSeries server to test that we have setup the Java environment on the iSeries server correctly. Follow these steps to compile Java source code, run a Java program, and create a Java class:

1. Create the Java source code by following these steps:
  - a. Map a network drive to the specified integrated file system directory. For this example, we use *mydir* as the directory name.
  - b. To open Notepad and create a new Java source named "HelloAS400World.java", enter:  
`notepad HelloAS400World.java`
  - c. Enter the source code shown in Figure 2-17.

**Note:** Use the exact punctuation and capitalization as shown here. Both are critical to the Java syntax.

```
public class HelloAS400World  
{  
    public static void main(String args[])  
    {  
        System.out.println("Hello World from AS/400!");  
    }  
}
```

Figure 2-17 HelloAS400World program

- d. Exit and save to your directory on the iSeries server.
2. Compile the source by completing these tasks:
- a. Enter **qsh** on the OS/400 command line. Press Enter.  
You are now in the Qshell environment, which is like a UNIX environment.
  - b. Enter **cd mydir** to access your directory. Press Enter.
  - c. Try using a few Qshell commands, such as: **ls** (list files, like **dir**), **env** (show environment), **find Hello\*.\*** (similar to **ls**), **hostname**, and so on.
  - d. On the QSH Command Entry command line to compile the program, as shown in Figure 2-18, enter:  

```
javac HelloAS400World.java
```

Press Enter.  
The \$ symbol signifies the end of the compile. If there are any errors in the source, they are printed to the display.

QSH Command Entry

```
$  
> cd mydir  
$  
> javac HelloAS400World.java  
$  
  
====>  
  
F3=Exit F6=Print F9=Retrieve F12=Disconnect  
F13=Clear F17=Top F18=Bottom F21=CL command entry
```

*Figure 2-18 Compiling a Java program using the Qshell Interpreter*

If successful, a file called HelloAS400World.class is created inside of the directory in the integrated file system. You can look for this file from the DOS prompt or Qshell environment. You can also look for it by using the Work with Object Links (WRKLNK) CL command.

3. Run the HelloAS400World application by performing these steps:
- a. On the QSH Command Entry display to see a list of valid parameters for the **java** command, enter:  

```
java -help
```

Press Enter.
  - b. To see which JDK level is loaded on the system, enter:  

```
java -version
```

In this example, we are using JDK 1.2.2. Press Enter.
  - c. To run the application, enter:  

```
java -classpath /mydir HelloAS400World
```

This message Hello World from AS/400! should be displayed. See Figure 2-19.

```

QSH Command Entry

> java -version
java version "1.2.2"
$
> java -classpath /mydir HelloAS400World
Hello World from AS/400!
$

====>

F3=Exit F6=Print F9=Retrieve F12=Disconnect
F13=Clear F17=Top F18=Bottom F21=CL command entry

```

*Figure 2-19 Running a Java program using the Qshell Interpreter*

d. Try running it again, but this time enter:

```
java -verbose -classpath /mydir HelloAS400World
```

Press Enter.

This shows all of the classes as they are loaded by the JVM class loader. This is very helpful when you are trying to determine problems, especially with the CLASSPATH.

You have now successfully run a Java application using the iSeries JVM.

## 2.2.5 Setting the QUTCOFFSET system value

The QUTCOFFSET system value indicates differences in hours and minutes between Universal Time Coordinated (UTC), also known as Greenwich Mean Time (GMT), and the current system time (local time). This is the number of hours that you need to subtract from the current system time (local time) to obtain the UTC.

The JVM uses this value to keep track internally of the time in your system locale. Java has the ability to track time in other locations using locales. This example (the JVM uses the UTC) derives the correct date (and time):

```

import java.text.*;
import java.util.*;
import java.util.Date;

public class DateExample {

    public static void main(String args[]) {

        // Get the Date
        date now = new Date();

```

```
// Get date formatters for default, German locales
DateFormat theDate = DateFormat.getDateInstance(DateFormat.LONG);
DateFormat germanDate = DateFormat.getDateInstance(DateFormat.LONG,
    Locale.GERMANY);
```

For more information on the QUTCOFFSET system value, see Chapter 2 in *OS/400 Work Management*, SC41-5306.

## 2.3 Installing the IBM Toolbox for Java on the workstation

This section shows you how to install the IBM Toolbox for Java classes on the workstation. Although this is not mandatory, you may want to install the IBM Toolbox for Java on your workstation if you want to build a client/server application. This set of Java APIs provides easy-to-use Java classes to access most iSeries resources from Java applets or applications running on any Java-enabled workstation.

The IBM Toolbox for Java provides the following support:

- ▶ **Database**
  - *JDBC*: DB2 UDB for iSeries data can be accessed using a JDBC driver written to the interface defined by the JDBC 2.0 specification.
  - *Record-Level database access*: Physical and logical files on the server can be accessed a record at a time using the interface of these classes. Files and members can be created, read, deleted, and updated.
- ▶ **Integrated file system (IFS)**: The file system classes allow access to files in the IFS of the server. Through the IFS file classes, a Java program can open an input or output stream, open a file for random access, list the contents of a directory, and do other common file system tasks.
- ▶ **Programs**: Any iSeries or AS/400 program can be called. Parameters can be passed to the server program, and data can be returned to the Java program.
- ▶ **Commands**: Any non-interactive OS/400 command can be run. A list of OS/400 messages generated by the command is available when the command completes.
- ▶ **Data queues**: Access to both keyed and sequential data queues is provided. Entries can be added to and removed from a data queue, and data queues can be created or deleted on the server.
- ▶ **Print**: Access to server print resources is provided. Using the print classes, you can retrieve lists of spooled files, output queues, printers, and other print resources. You can work with output queues and spooled files, answer messages for spooled files, and do other print-related tasks. In addition, classes are provided to create new spooled files on the server and to generate SCS printer data streams. Writing directly to these classes, applications and applets can generate output on the server spool system.
- ▶ **Jobs**: Access server jobs and job logs is available. Using the job classes, you can retrieve messages in a job log, information about a job (name, number, type, user, status, job queue, and more), or obtain a list of jobs based on your selection.
- ▶ **Messages**: Access OS/400 messages, message queues, and message files is offered. Using the message classes, you can retrieve a message that is generated from a previous operation (such as a command call) or information about a message on a message queue. You can also interact with a message queue that allows you to send, receive, and even reply to messages.

- ▶ **Users and groups:** Access users and groups is provided. The user and group classes allow to you obtain a list of users and groups on the server and information about each user.
- ▶ **User spaces:** You can access user spaces. Use the user space class to read, write, create, and delete user spaces on the server.
- ▶ **Data areas:** You can access various kinds of data areas (character, decimal, local, logical). Use the data area classes to read, write, create, and delete data areas on the server.
- ▶ **System values:** Query and reset system values and network attributes on the server.
- ▶ **System status:** Retrieve system status information. Using the SystemStatus class, you can retrieve such information as the total number of user jobs and system jobs currently running on the server and the storage capacity of the server's auxiliary storage pool.

The IBM Toolbox for Java is shipped as a set of JAR files. Each JAR file contains Java packages that provide specific functions:

- ▶ **jt400.jar:** Access, resource, vaccess, security, and PCML classes and JDBC support.
- ▶ **jt400.zip:** Access, resource, vaccess, security, and PCML classes (the same classes as contained in jt400.jar). The jt400.zip file is shipped to retain compatibility with previous releases of IBM Toolbox for Java. Use jt400.jar instead of jt400.zip.
- ▶ **jt400Access.zip:** Access, resource, security, and PCML classes (that is, the same classes that are in jt400.jar minus the vaccess classes). The jtAccess400.zip is shipped to retain compatibility with previous releases of IBM Toolbox for Java. Use jt400.jar or jt400Native.jar instead of jt400Access.zip.
- ▶ **jt400Native.jar:** Access, resource, security, PCML, and native optimizations, which is a set of classes (less than 20) that take advantage of the iSeries functions when it runs on the iSeries JVM. Because jt400Native.jar contains the native optimizations, when running on the iSeries JVM, use jt400Native.jar instead of jt400.jar. The jt400Native.jar ships with OS/400 and resides in the /QIBM/ProdData/OS400/jt400/lib directory.
- ▶ **jt400Native11x.jar:** Native optimizations only. If you are running on the iSeries JVM and want to use jt400.jar, include jt400Native11x.jar in the CLASSPATH instead of jt400Native.jar. The jt400Native11x.jar ships with OS/400 and resides in the /QIBM/ProdData/OS400/jt400/lib directory.

You can choose to use the IBM Toolbox for Java classes directly from the iSeries IFS by setting the CLASSPATH variable, or you can copy the IBM Toolbox for Java classes on the workstation hard disk drive.

### 2.3.1 Setting up the CLASSPATH variable

To directly use the IBM Toolbox for Java classes, you simply need to map a network drive on the workstation to the iSeries IFS and direct the JVM on the workstation to this drive with an appropriate CLASSPATH directive.

When working on Windows 98, you do this by using a CLASSPATH directive that you add to the autoexec.bat file.

To change the autoexec.bat file, you need to complete these steps:

1. Start a text editor, such as WordPad or Notepad.
2. From the File menu, open **c:\autoexec.bat**.
3. Look at the **SET CLASSPATH=** statement.

**Note:** The CLASSPATH statement is a series of directories that are separated by semicolons (;). The JVM looks for Java classes in the CLASSPATH directories in order, from left to right.

4. If a SET CLASSPATH= statement does not exist, insert a new one.
5. Place the IBM Toolbox for Java directory at the end of the SET CLASSPATH= statement.
6. Choose **Save** from the **File** menu to update the autoexec.bat file.
7. Exit from the text editor.

For example, on these SET statements, we set up the IBM Toolbox for Java directory structure:

```
SET AS400JT=S:\QIBM\ProdData\HTTP\Public\jt400
SET CLASSPATH=.;%AS400JT%\lib\jt400.jar;
```

First, you set your own environment variable, called AS400JT, with the value of the IBM Toolbox for Java main directory on the integrated file system. This assumes that you have mapped a network drive "S:" to the iSeries server.

Then, you set the CLASSPATH environment variable to instruct the JVM that the IBM Toolbox for Java classes are contained in the jt400.jar file that is located in the LIB subdirectory. Setting up your own environment variable allows for easier changes later if you decide to download the IBM Toolbox for Java classes on the hard disk of the workstation.

Once you add the proper path information to the autoexec.bat file, you must run it for the new path information to take effect.

You should re-boot your system. However, you system can run interactively (use care as to what else will run again) by entering these commands at the DOS prompt:

```
C:\WINDOWS>cd \
C:>autoexec.bat
```

This completes setting up the Java environment on the workstation.

When working on Windows NT/2000, you set the CLASSPATH by following these steps:

1. Right-click the **My Computer** icon on your desktop, and select **Properties**.
2. On the System Properties display that appears, click the **Environment** tab. It shows the variables both system-wide and user level as shown in Figure 2-20 on page 56.

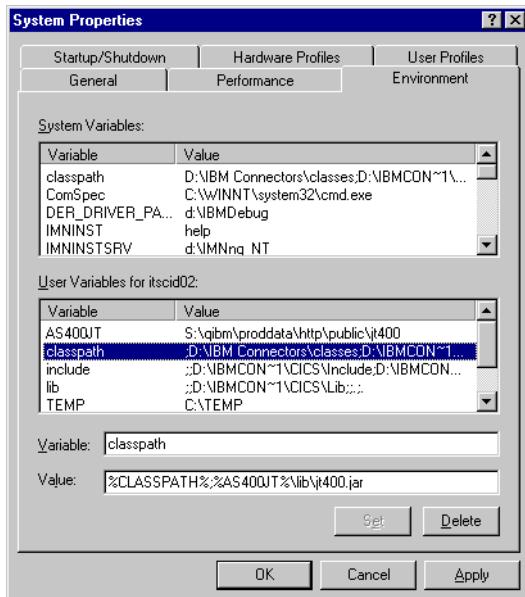


Figure 2-20 System Properties

3. You can choose to set either the system level or the user level. In our example, we set the user level. First we create a variable AS400JT as shown in Figure 2-21.



Figure 2-21 Setting the AS400JT variable

4. If the CLASSPATH variable does not exist, create a new one. We set the CLASSPATH as shown in Figure 2-22. The %classpath% represents the CLASSPATH variable at the system level. The directories are separated by a semi-colon (;).

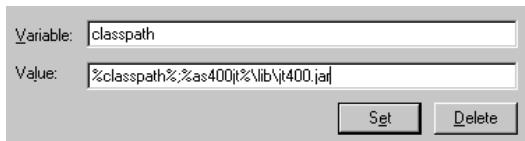


Figure 2-22 Setting the CLASSPATH variable

This is the easiest way to use the IBM Toolbox for Java classes. It is also the only way to use these classes if you are running Java on a diskless device, such as the IBM Network Station. As with any other iSeries LPP, changes to the IBM Toolbox for Java are distributed by means of PTFs. PTFs are applied to the iSeries server. Using the classes directly from the iSeries integrated file system ensures that you automatically have all of the changes after they are applied to the iSeries server.

### 2.3.2 Copying the IBM Toolbox for Java classes to the workstation

For performance reasons, you may prefer to copy the IBM Toolbox for Java classes to your workstation hard disk drive. You can reduce the amount of required storage space by using only the JAR files required to enable the specific functions that you want.

To copy the files from the iSeries server to the workstation, perform these steps:

1. Decide which method you want to use to copy the files to the workstation. You can either use the AS400ToolboxInstaller class or manually copy either the ZIP or JAR file. Perform these tasks:
  - a. The IBM Toolbox for Java information fully documents the AS400ToolboxInstaller class. In the IBM Toolbox for Java information in the iSeries Information Center, look under **Tips for programming** and then **Install and update**. This class is located in /QIBM/Proddata/HTTP/Public/jt400/utilities.
  - b. Find the file named jt400.jar. It should reside in the /QIBM/ProdData/HTTP/Public/jt400/lib directory. Copy the **jt400.jar** file from the iSeries server to the workstation. You can do this in a variety of ways. The easiest way is to use Client Access to map a network drive on the workstation to the iSeries server. You can also use file transfer protocol (FTP) to send the file to the workstation. If you do this, make sure that you transfer the file in binary mode.
2. Update the CLASSPATH environment variable of the workstation by adding the location where you put the program files. For example, on a personal computer that is using Windows 98 operating system, if jt400.jar resides in C:\jt400\lib\, append **;C:\jt400\lib\jt400.jar** to the CLASSPATH.

To update the CLASSPATH, you need to change the CLASSPATH environment variable by modifying:

- ▶ The autoexec.bat file on Windows 98
- ▶ The system properties on Windows NT/2000

Refer to 2.3.1, “Setting up the CLASSPATH variable” on page 54. Now the JVM on the workstation loads the IBM Toolbox for Java classes from the workstation hard disk, instead of loading them from the iSeries integrated file system.

You are now ready to use Java and the IBM Toolbox for Java classes on the workstation and on the iSeries server.

## 2.4 Using Remote AWT support on the workstation

The Remote Abstract Windowing Toolkit is a set of Java classes that handle graphical user interface (GUI) operations on a server, which does not have any GUI capable devices directly attached to it, such as the iSeries server. The Remote AWT feature of the IBM Developer Kit for Java provides an easy way to test and run (on the iSeries server) any Java application that uses the Java AWT or Swing classes.

Typically, the only GUI operations that are performed on the server side of a client/server Java application are those that are related to application installation and configuration. This includes application functions that must be performed on the server and require limited user interaction. It is expected that all GUI-intensive operations are performed on the client side where the user interaction with the application takes place.

### 2.4.1 Setting up the Remote AWT environment

This section shows you how to use Remote AWT with V5R1. To setup the Remote AWT environment, make sure that you have downloaded and installed the Sun Microsystems, Inc. Java Development Kit (JDK) on the workstation. Then, setup access to Remote AWT by accessing the RAWTGui file using either of these methods:

- ▶ The first method is explained here:
  - a. Map a network drive on the workstation to the iSeries integrated file system, and set the proper CLASSPATH environment variable. The JVM that runs on the workstation must be able to find the Remote AWT classes that are contained in:
    - For JDK 1.1.8, the RAWTGui.zip file that is located in the /QIBM/ProdData/Java400/jdk118 directory in the integrated file system
    - For JDK 1.2, the RAWTGui.jar file that is located in the /QIBM/ProdData/Java400/jdk12 directory in the integrated file system
    - For JDK 1.3, the RAWTGui.jar file that is located in the /QIBM/ProdData/Java400/jdk13 directory in the integrated file system
  - b. Update the CLASSPATH statement for Remote AWT by adding the RAWTGui file to your environment. To do this, enter the following statement on a line in the autoexec.bat file for Windows 98 or the system properties for Windows NT/2000:
 

```
SET CLASSPATH=.;S:\QIBM\ProdData\Java400\jdk118\RAWTGui.zip
```
- ▶ If you do not want to run using a network drive, use this method:
 

Copy the RAWTGui file to the workstation. You can use a network drive or FTP to do this. Create a directory on the workstation (for example rawt). Then, copy the Remote AWT RAWTGui file from the iSeries integrated file system to the workstation.

Then, update the CLASSPATH to find the Remote AWT classes. The path should look like this:

```
SET CLASSPATH=C:\rawt\RAWTGui.zip
```

You are now ready to start Remote AWT.

## 2.4.2 Starting Remote AWT support on the workstation

Start the Remote AWT daemon on the workstation, so it listens on a TCP/IP port for incoming Remote AWT requests that are sent by the iSeries Java application.

To do this, open a command prompt.

- ▶ For JDK 1.1.x, at the prompt, enter:

```
java com.ibm.rawt.server.RAWTPCServer
```

Press Enter. As shown in Figure 2-23, the Welcome to Remote-AWT message window appears, which includes the Remote AWT version number and the workstation IP address.



Figure 2-23 Remote AWT welcome message

- ▶ For JDK 1.2 and higher, at the prompt, enter:

```
java -jar RAWTGui.jar
```

Press Enter. As shown in Figure 2-24, the Remote AWT for Java 2 window appears, which includes the Remote AWT version number and the workstation IP address.



Figure 2-24 Remote AWT daemon

If you want to automate the entire process, you can create a .bat file and run it when you want to start the Remote AWT support on the workstation.

To create a .bat file, follow these steps:

1. Start an editor, such as WordPad or Notepad.
2. Type the SET CLASSPATH and the **start /m** statements as shown here:  

```
SET CLASSPATH=C:\rawt\RAWTGui.zip
start /m java com.ibm.rawt.server.RAWTPCServer
```
3. Save the file in the folder, and give it a meaningful name such as *rmtawt.bat*.
4. Exit from the editor.

Now, you can start the Remote AWT daemon on the workstation. Enter **rmtawt** at the DOS prompt whenever you need to run a Java application on the iSeries server that uses the Java AWT APIs. You may even want to include this in the autoexec.bat file to have the daemon ready on the workstation.

You do not have to restart Remote AWT on the remote display after the Java application program ends. The RAWT daemon selects the first free port above 2000 when the Java application connects using Remote AWT. The Java application uses this port until the application ends. Additional Java applications are connected to subsequent free ports above 2000. The available range of ports goes up to 9999.

### 2.4.3 Running Remote AWT support on the iSeries server

Use the RUNJAVA (or JAVA) command to start the application. To use this method, set up the Java properties as described here:

1. Enter the Run Java (**RUNJAVA**) command on the command line.  
**Note:** You must define the Java classpath for the Java program.
2. Press the F4 (Prompt) key to display the command prompt.
3. Enter the Java program class name that you want to run on the class parameter line.
4. Press the F10 (Additional parameters) key for more prompting.
5. Press the Page Down key to go to the next page of the prompt.
6. Enter **awt.toolkit** on the property name parameter line if it is not already entered as the default.
7. Enter **com.ibm.rawt.client.CToolkit** on the property value parameter line if it is not already entered as the default.
8. Enter + for more properties. The display should now appear as shown in Figure 2-25 on page 60.

```

Run Java Program (RUNJVA)

Type choices, press Enter.

Optimization . . . . . 10      10, *INTERPRET, 20, 30, 40
Interpret . . . . . *OPTIMIZE *OPTIMIZE, *YES, *NO
Properties:
  Property name . . . . . awt.toolkit

  Property value . . . . . com.ibm.rawt.client.CToolkit

  + for more values +
Garbage collect initial size . . 2048      256-139264000 kilobytes
Garbage collect maximum size . . *NOMAX    256-139264000 kilobytes
Garbage collection frequency . . 50        0-100
Garbage collection priority . . 20        20, 10, 30
Option . . . . . *NONE    *NONE, *VERBOSE, *DEBUG...
  + for more values

Bottom
F3=Exit   F4=Prompt   F5=Refresh   F12=Cancel   F13=How to use this display
F24=More keys

```

Figure 2-25 Using the Run Java Program (RUNJVA) command to Run MyApp

9. Enter **RmtAwtServer** on the next property name parameter line.
10. Enter the Transmission Control Protocol/Internet Protocol (TCP/IP) address (for example, 1.1.11.11) of the Remote AWT workstation.
11. Enter **os400.class.path.rawt** on the property name parameter line.
12. Enter **1** on the property value parameter line.

The command line should look like this:

```
java class(classname) prop((awt.toolkit com.ibm.rawt.client.CToolkit) (RmtAwtServer '1.1.11.11') (os400.class.path.rawt 1))
```

**Note:** Starting with V4R4, the awt.toolkit property name defaults to the com.ibm.rawt.client.CToolkit property value, so this property is not required in the command line. Therefore, the command line looks like this:

```
java class(classname) prop((RmtAwtServer '1.1.11.11') (os400.class.path.rawt 1))
```

13. Press the Enter key to start the application.

## Multiple JDK versions

If you have multiple JDK versions installed, follow step 1 through step 13 as described in 2.4.3, “Running Remote AWT support on the iSeries server” on page 59. Then, add this property to select the JDK version if it is different than the current version:

1. Enter **java.version** on the property name parameter line.
2. Enter **x.x.x** on the property value parameter line. The command line should look like this:

```
java class (classname) prop((RmtAwtServer '1.1.11.11') (os400.class.path.rawt 1)(java version 1.3))
```

3. Press Enter.

Again, you may choose to specify the workstation host name rather than its IP address. In this case, make sure the names are set up properly in the iSeries host table and the Domain Name Server and Windows workstation TCP/IP properties.

In each of the preceding examples, a small application named MyApp was used to test the setup. This is a simple Java application that displays a simple panel on the workstation. The source appears as shown in the following example:

```
import java.awt.*;
public class MyApp extends Frame
{
    // "final" variables are constants
    static final int H_SIZE = 300;
    static final int V_SIZE = 200;
    public MyApp()
    {
        // Calls the parent constructor
        // Frame(string title)
        // Equivalent to setTitle("My First Application")
        super("My First Application");
        pack();
        setSize(H_SIZE, V_SIZE);
        show();
    }
    public static void main(String args[])
    {
        new MyApp();
    }
}
```





# Introduction to VisualAge for Java

VisualAge for Java is an integrated, visual environment that supports the complete cycle of Java program development. You can create Java applets, which run in Web browsers, and stand-alone Java applications. You can also create server-side Java support like servlets and Enterprise JavaBeans.

This chapter provides an introduction to VisualAge for Java with a specific emphasis on the components most likely to be used by an iSeries development team. It discusses the following topics:

- ▶ VisualAge Java overview
- ▶ Integrated Development Environment (IDE)
  - Java support
  - Navigating within VisualAge for Java
  - Visual Composition Editor (VCE)
  - Team development
  - Applet viewer
  - Editor/Debugger/SmartGuides
- ▶ System requirements and prerequisites

## 3.1 VisualAge for Java overview

IBM VisualAge for Java is one of the first enterprise-wide, team-enabled, incremental application development environments for Java in the industry. It is designed to connect Java clients to existing server data, transactions, and applications. This enables developers to extend server-based applications to communicate with Java clients on the Internet or intranet, rather than rewrite the application from scratch. VisualAge for Java creates 100% pure Java-compatible applications, applets, and JavaBeans.

This chapter discusses various processes and windows that you use in the development of applications using VisualAge for Java. All development for this Redbook was performed using the Windows NT client of the Enterprise Edition of VisualAge for Java Version 3.5.3. If you are using a different client or the Professional Edition, there may be some slight differences in the processes and windows that are discussed and shown here.

The source code for any of the examples discussed in this chapter is available on the Internet. For download instructions, please refer to Appendix A, "Additional material" on page 529.

### 3.1.1 VisualAge for Java versions

VisualAge for Java is available in four scalable packages:

► **Entry Edition**

VisualAge for Java, Entry Edition, is suitable for learning purposes and for building small projects. It is available as a no-charge download from VisualAge Developer Domain at <http://www.ibm.com/software/vadd>

Two free, scaled-down versions of products, Entry Professional Edition for Windows and Entry Enterprise Edition for Windows, are available. They are limited to holding up to 750 classes in the workspace. IBM doesn't provide support for these trial products.

Documentation can be viewed through the Support page on the Web. The user is licensed for "internal, non-commercial use". Note these points:

- Users must be registered in the VisualAge Developer Domain. For information, see: <http://www7.software.ibm.com/vad.nsf>
- No support provided
- 750-class limit
- Does not support the IBM Toolbox for Java

► **Professional Edition**

VisualAge for Java, Professional Edition, is a complete Java development environment that includes easy access to JDBC-enabled databases for building full-function Java applications. It includes an editor, debugger, browser, and Visual Composition Editor. The Professional Edition is included with the VisualAge Developer Domain Subscription for Java. Note these advantages:

- Provides IDE and Visual Composition Editor (VCE)
- Provides JSP/Servlet Development Environment including a Create Servlet SmartGuide, which optimizes generated servlets, HTML, and JSP components
- Includes the WebSphere Test Environment
- Includes the XML Parser for Java
- Includes Data Access Beans, DB2 Stored Procedure Builder and SQLJ, to access data
- Provides Distributed Debugger and Integrated Debugger to debug your applications
- Supports the IBM Toolbox for Java
- Supports template for Customer Relationship Management (CRM)
- Allows non-Java files to be managed in the repository
- Includes Integration with SCM systems (TeamConnection, ClearCase, and PVCS)

► **Professional Edition shipped with WebSphere Development Studio for iSeries**

For OS/400 V5R1, the WebSphere Development Studio for iSeries (5722-WDS) is available. It has two groups of components, Host Component and Workstation Component. The Workstation Component includes a customized version of VisualAge for Java for iSeries 3.5 (Professional Edition), which provides easy access to iSeries server functions and generates code to access iSeries data and applications.

For V5R1, WebSphere Development Tools for iSeries (5724-A81) is also available. It includes only the workstation components, including VisualAge for Java for iSeries.

For V4R5, 5769-WDS is also available. It contains an unlimited license of 5724-A81.

► **Enterprise Edition**

Enterprise is a superset of Professional Edition. VisualAge for Java, Enterprise Edition, is designed for building enterprise Java applications. It has all of the Professional Edition features plus support for developers who work in large teams, who develop high-performance or heterogeneous applications, or who need to connect Java programs to existing enterprise systems.

It includes these features:

- All Professional Edition support
- XMI Toolkit provides a bridge between the VisualAge for Java and Rational Rose tool
- XML Generator can edit a DTD and generate sample XML documents based on a DTD
- EJB Development Environment enables you to develop beans that implement Sun EJB programming specifications
- IDL Development Environment, and Persistence Builder, C++ Access Builder
- Enterprise Access Builder (EAB) for Transactions
- Domino AgentRunner, Domino Access Builder, Access Builder for SAP R/3
- Enterprise Toolkit for iSeries
- Enterprise Toolkit for S/390
- Team Programming provides an integrated team development environment
- Tool Integrated API, Remote Access to Tool API, Tivoli Connection

Beyond the current batch-based Java tools available today, VisualAge for Java provides:

- Superior enterprise connectivity
- Project-based team development
- A true incremental *rapid application development* environment for Java

VisualAge for Java is part of the VisualAge family of products and shares some of the components from the other VisualAge products. For example, VisualAge for Java shares the team environment repository and image concepts (and implementation) with the VisualAge for SmallTalk product. It also shares the VCE component, which is common across all development environments.

### **3.1.2 Integrated Development Environment**

The Integrated Development Environment (IDE) that is incorporated within the product enables the developer to code, compile, test, and debug single lines of code, as well as full-scale applications, enabling the application to scale with the business requirement. The IDE is built around the industry leading ENVY/Developer team development environment from OTI (an IBM subsidiary company). It is well recognized within the object technology marketplace for its ability to provide management facilities for small and large scale application development projects.

The IDE enables a developer to build and run applications, applets, and code snippets interactively without needing to run the compile statement (`javac`) from the command line. All applications can be run from within the IDE without exporting the Java source or class files. This is achieved through the provision of a JDK 1.2 compliant virtual machine (VM) within the IDE. Because you can interactively modify code and run it without compilation, developers can debug code on the fly. They can spot errors in their code with the debugger, change it, and continue without bringing the running application down – all within the VisualAge for Java IDE.

VisualAge for Java is an open IDE. Developers can easily import and export Java source and class files, as well as JavaBeans that may have been purchased by the company or made available on the Internet. The JavaBeans support in VisualAge for Java also enables a developer to import an existing JavaBean (for example, from the Internet) into VisualAge for Java, modify the bean, and export it again for use within another JDK 1.2 compliant development environment (for example, Symantic Cafe and Borland JBuilder).

Version 1 of VisualAge for Java supports JDK 1.1.4. Version 2 supports JDK 1.1.7, Version 3.5 supports JDK 1.2.2. Along with the current JDK support, VisualAge for Java also supports all the most current standards for Java development (for example, Java Database Connectivity (JDBC) and so forth), which is discussed later. Because of the portability of JDK 1.2 compliant Java code, code that is developed using VisualAge for Java can run without being changed on the native iSeries Java virtual machine.

VisualAge for Java has several components that extend its capabilities to make client/server programming easier. The Enterprise Access Builders for Transaction provide components to aid connection to existing transaction-based systems, and Data Access Bean helps to access relational database. In addition, the IBM Toolbox for Java provides a series of classes specifically designed to access many iSeries features (all without using IBM Client Access as a prerequisite).

## 3.2 Using the Integrated Development Environment

This section introduces the fundamental elements of the VisualAge for Java IDE that are accessed from the Workbench window in the IDE. It covers:

- ▶ Starting VisualAge for Java
- ▶ The Workbench and its hierarchy:
  - Projects
  - Packages
  - Resources
  - Classes
  - Interfaces
  - Managing
  - All problems
- ▶ Browsers:
  - Project
  - Package
  - Class

### Starting VisualAge for Java

During the installation of VisualAge for Java, an item is added to the Windows taskbar — *IBM VisualAge for Java for Windows V3.5*. This item has a number of sub-items, and selecting IBM VisualAge for Java starts VisualAge for Java (Figure 3-1).

During the startup process, VisualAge for Java loads the development image. Since this image can be 8 MB or larger (typically in the 15 MB to 25 MB range), the startup process can take one to two minutes because the entire image must be loaded into memory. The development image is also known as the *workspace*. These two terms are used interchangeably in this chapter. If this is the first time VisualAge for Java is started, the first window displayed is the Welcome dialog (see Figure 3-2).

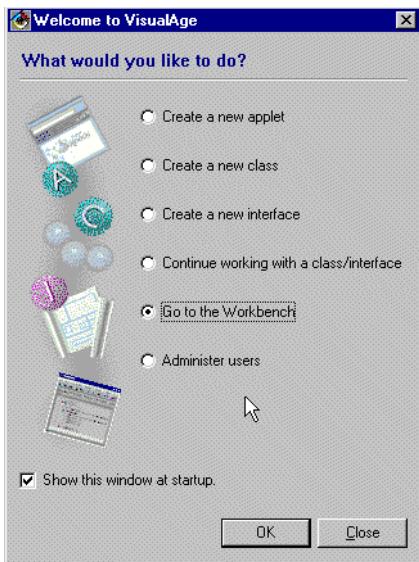


Figure 3-2 Welcome dialog

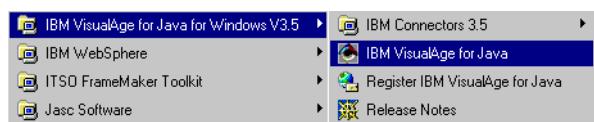


Figure 3-1 Starting VisualAge for Java

The Welcome dialog window provides a single point to perform most of the simple tasks. However, as you gain more experience using VisualAge for Java, you may decide to stop this window from appearing at startup. To do so, click the corresponding check box. If you want to reactivate the Welcome dialog later, refer to Figure 3-68 on page 115.

Select **Go to the Workbench** and click **OK** to go to the Workbench window. The window shown in Figure 3-3 on page 68 appears.

The Workbench is the main window into the workspace. You organize your work from the Workbench. From here, you can open several other windows to help with your tasks. As you open windows, navigate in them, create source code, and perform other tasks, the workspace is modified. From the Workbench, you can open specialized windows (called *browsers*) on individual program elements in the workspace.

The Workbench window is split into a number of areas that are common across most of the VisualAge for Java windows:

- ▶ **Title bar**  
Contains the title and current user.
- ▶ **Menu bar**  
Provides access to all functions.
- ▶ **Tool bar**  
Provides fast access to most used menu items.
- ▶ **Tool tip**  
Shows the meaning of the toolbar buttons.
- ▶ **Notebook tab**  
Provides a view of the five fundamental components of VisualAge for Java (projects, packages, resource, classes, and interfaces), as well as a tab for managing them and one for displaying any unresolved problems.
- ▶ **Hierarchy pane**  
Typically displays the component being browsed in context with its contained components. For example, a project browser shows all of its packages/resources. Each package is expandable to show all of the classes/interfaces it contains and so forth.

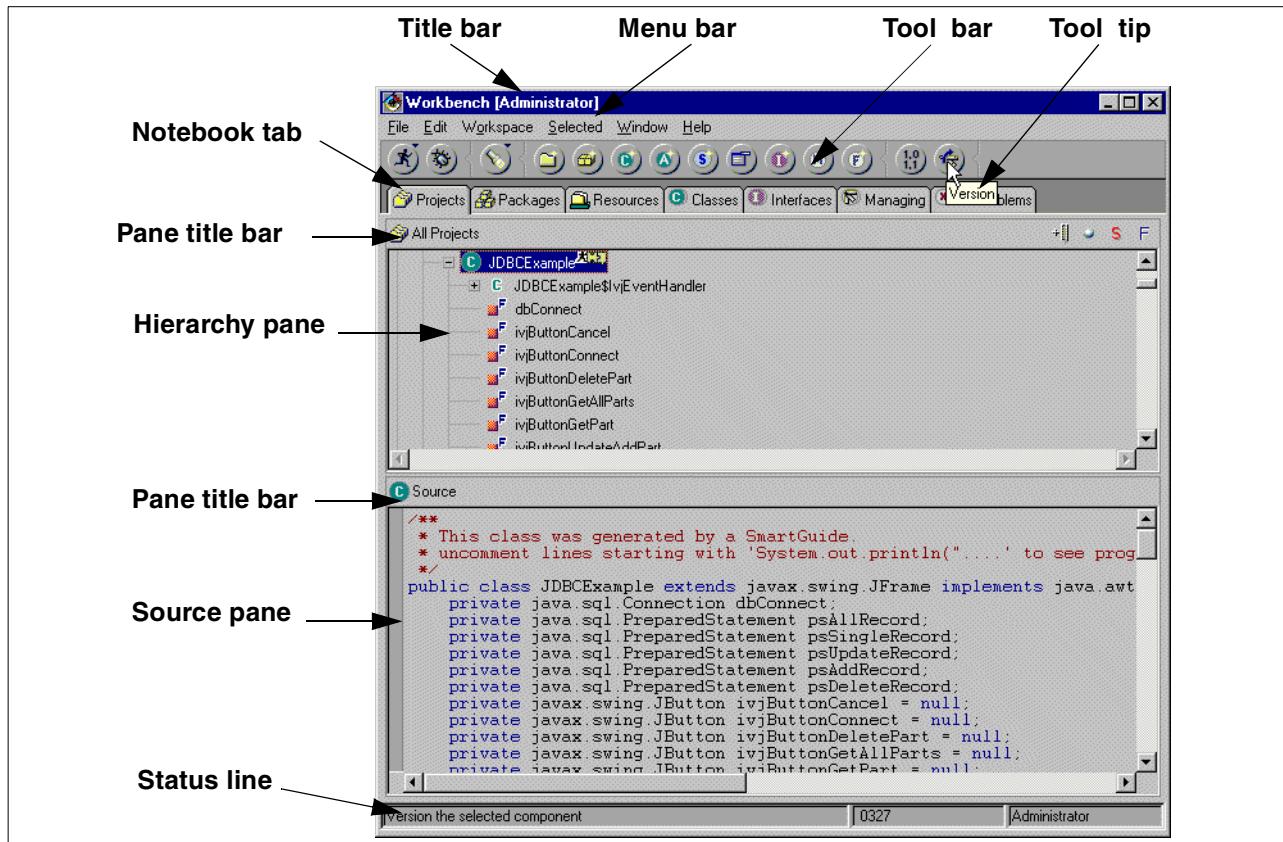


Figure 3-3 VisualAge for Java Workbench

► **Source pane**

If a method is highlighted in the hierarchy pane, the method source code is displayed in the source pane. Similarly, if a class/interface is highlighted in the hierarchy pane, the class/interface definition is displayed in the source pane. Any source code can be edited directly in the source pane.

► **Pane title bar**

Maximizes or resizes the corresponding pane when double-clicked (works with any pane in VisualAge for Java).

► **Status line**

Provides feedback to the user on the current action, mouse position, selection, and so forth.

## Component hierarchy

Source code is stored as structured objects in the following hierarchy of VisualAge program elements:

```

Solutions
  Projects
    Packages
      Types (Classes or Interfaces)
        Methods or constructors
  
```

You are probably already aware of the package, class or interface, and method or constructor components that are part of the standard Java language. In addition, VisualAge for Java includes two higher grouping levels called *solutions* and *projects*. Solutions enable the grouping together of various *projects*, and projects enable the grouping together of various *packages*. Each higher level component can have multiple lower level components. For example, a project can contain one or more packages. VisualAge for Java also supports *resources*, which allow you to group non-Java resources file in the scope of a project.

Various icons are used in each of the browsers to depict each component. Examples of the icons used are shown in Figure 3-4 through Figure 3-9. Details about Icons used in VisualAge for Java can be found in the online help under **PDT Index-> Integrated Development Environment -> Chapter 7. IDE hints and tips-> VisualAge for Java IDE Symbols** (see “VisualAge for Java help” on page 119).

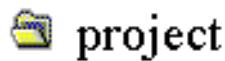


Figure 3-4 Project icon



Figure 3-5 Resource icon

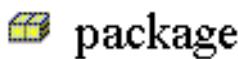


Figure 3-6 Package icon

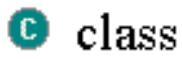


Figure 3-7 Class icon

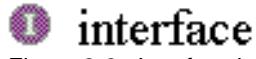


Figure 3-8 Interface icon



Figure 3-9 Executable class icon

## Workbench window

In the Workbench window (see Figure 3-10 on page 70), the IBM Toolbox for Java Example project is expanded to show its packages. One of these packages, the DatabaseAccessExample package, is expanded to show its classes and interfaces (classes only in this case). One of these classes, the DataQueueExample class, is expanded to show its methods. One of its methods, the `connectToDB (String, String, String)` method, is selected and its source is shown in the source pane.

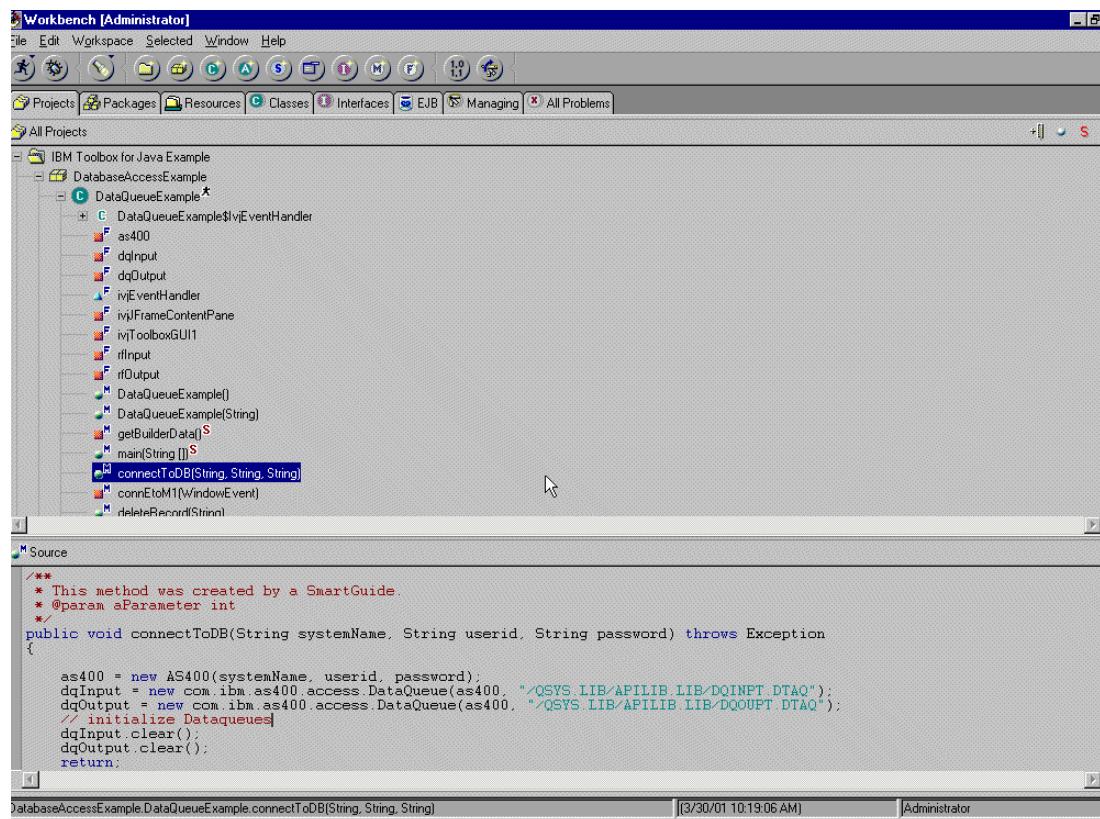


Figure 3-10 VisualAge for Java Workbench

## Component browsers



Figure 3-11 Browser selection

This section discusses the three component browsers used extensively within VisualAge for Java (projects, packages, and types, which includes classes and interfaces). You can open each of these browsers by selecting the menu **Workspace** from the menu bar. There, you select the menu item of the browser you want to open (see Figure 3-11).

You are prompted to choose one of the corresponding components, either project, package, or type. The browser appears after you make a selection from this window (see Figure 3-12).

Figure 3-12 shows the information and functions that the Component Selection window provides. They include:

- ▶ **Title bar**

Indicates which browser you are about to open.

- ▶ **Search line**

Provides a search including wildcards.

- ▶ **Component selection**

Displays all components from which to choose by clicking one of them.

#### ► Package selection

If a component exists in more than one package, you decide here from which of the packages the component should be taken. This applies only for types, since packages and projects must have unique names in your workspace.

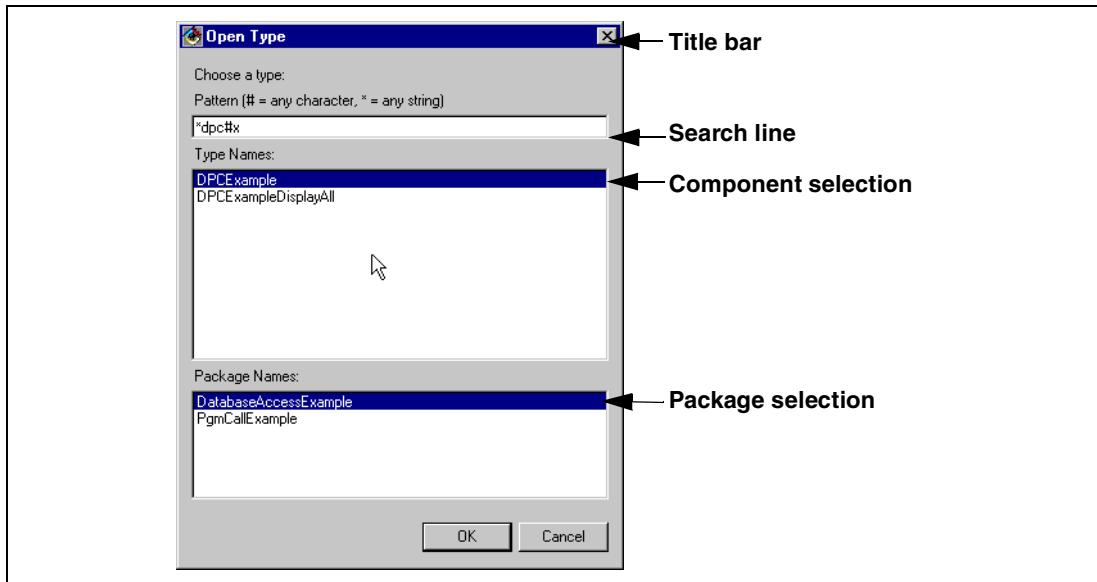


Figure 3-12 Component selection

Select the component and click **OK** to open the browser.

There are other ways to open a browser directly. For example, right-click a component shown on the hierarchy pane of the Workbench (see Figure 3-3 on page 68) and select **Open** from its pop-up menu. After that, the corresponding browser opens. Since Open is the default response when you double-click, you receive the same result.

## Project browser

The project browser displays details of all the components within the project, including the packages, resources, classes, interfaces, methods, comments, and source code. If you select a project, package, type, or method, you can display or edit the comments or source code directly in the source pane (see Figure 3-13 on page 72).

The project browser window has seven different views that you can access by clicking their notebook tabs:

#### ► Packages view

Provides detailed information about all packages, types, and methods within the project. All of this information can be displayed and edited in the source pane (Figure 3-13).

#### ► Resource view

Provide detailed information about resource files that are not Java source files or bytecode, such as HTML files, image files, audio clips, Stored Procedure Builder files, or SQLJ source files. Within this window, you can view a project's resource files, and you can also delete, rename, or open resource files.

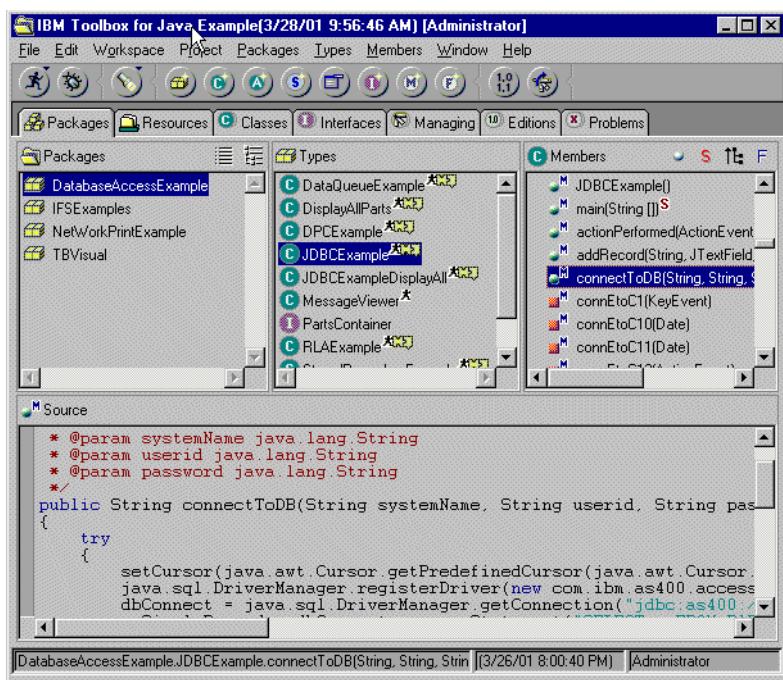


Figure 3-13 Project browser Packages view

#### ► Classes view

Provides detailed information about the class hierarchy, all classes, and their methods within the project (Figure 3-14).

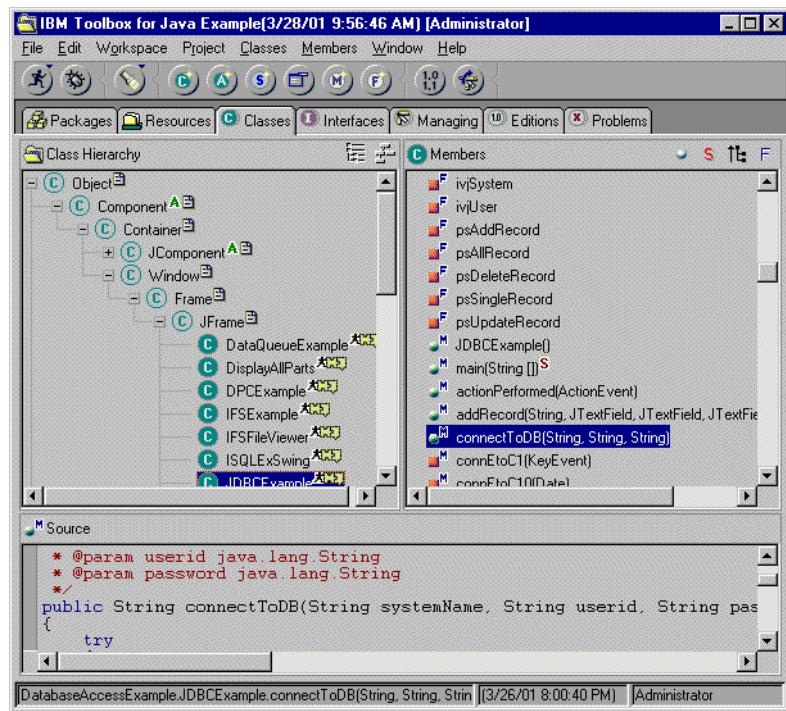


Figure 3-14 Project browser Classes view

#### ► Interfaces view

Provides detailed information about all interfaces and their methods within the project (Figure 3-15).

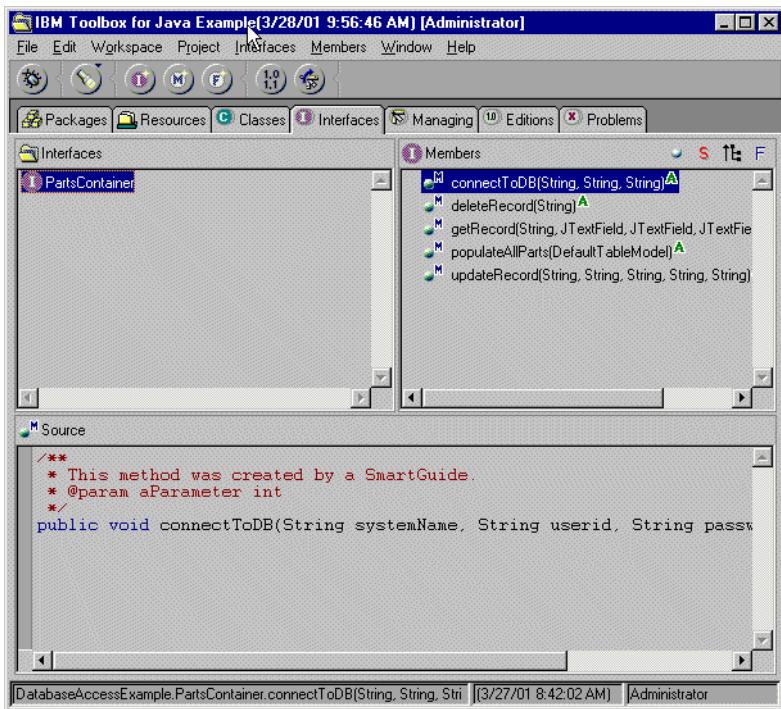


Figure 3-15 Project browser Interfaces view

#### ► Managing view

Provides detailed information about package group members (project team members) and ownership of types within the project. All of this information can be managed in the corresponding pane (Figure 3-16).

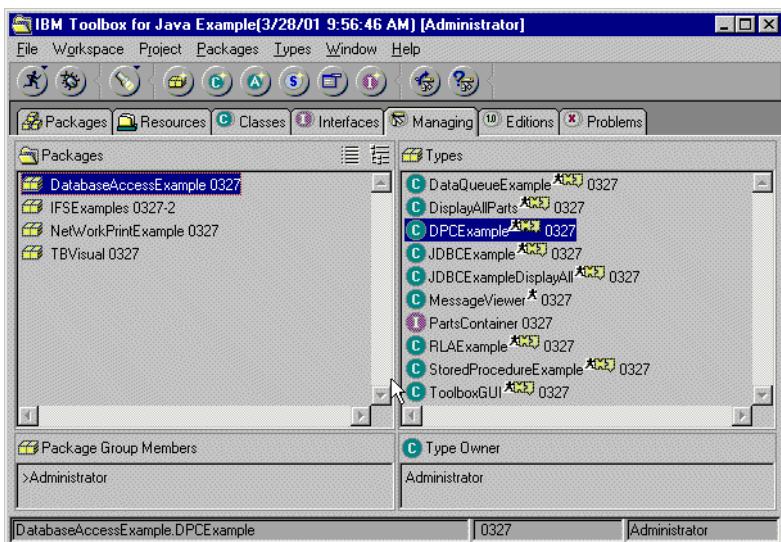


Figure 3-16 Project browser Managing view

#### ► Editions view

Provides detailed information about all versions and editions of packages, types, and methods within the project. It enables the developer to manage multiple versions and editions of packages, classes, interfaces, and methods (Figure 3-17).

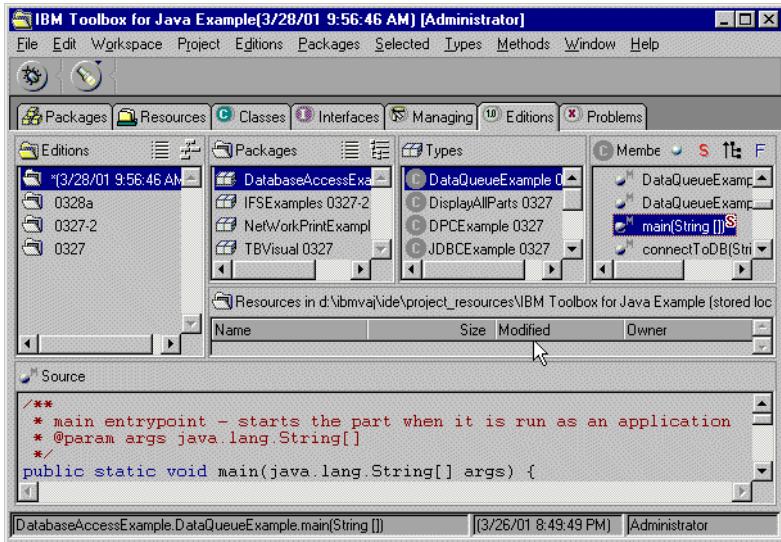


Figure 3-17 Project browser Editions view

#### ► Problems view

Provides detailed information about all unresolved problems within the project. All of this information can be managed in the corresponding pane (Figure 3-18).

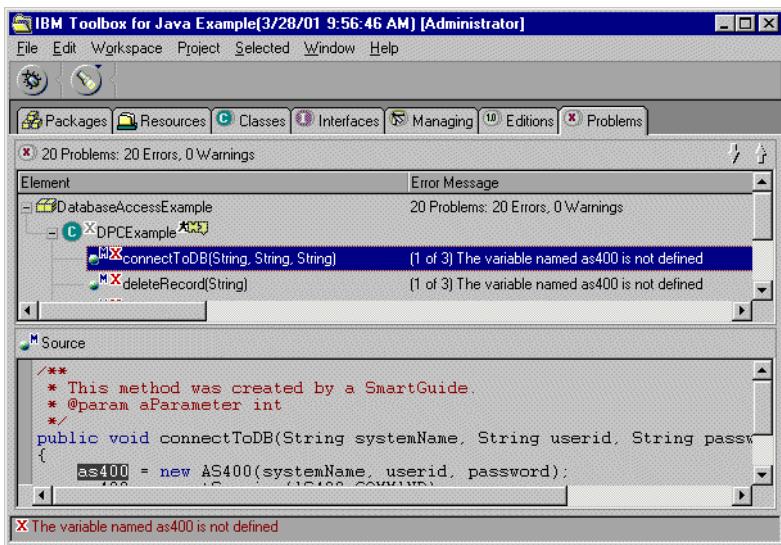


Figure 3-18 Project browser Problems view

## Package browser

The package browser displays details of all the components within the package, including the class hierarchy, classes, interfaces, methods, comments, and source code. If you select a package, type, or method, you can display or edit the comments or source code directly in the source pane. In the hierarchy pane, you can switch between a tree and graph layout.

The views in this browser have a slightly different layout, but they work basically the same way as the corresponding views of the project browser. The only difference is that you navigate within a package and not within a project. For this reason, the number of upper panes is reduced to two. There is more space for class and method names, and there's no Resource in the notebook tab (Figure 3-19).

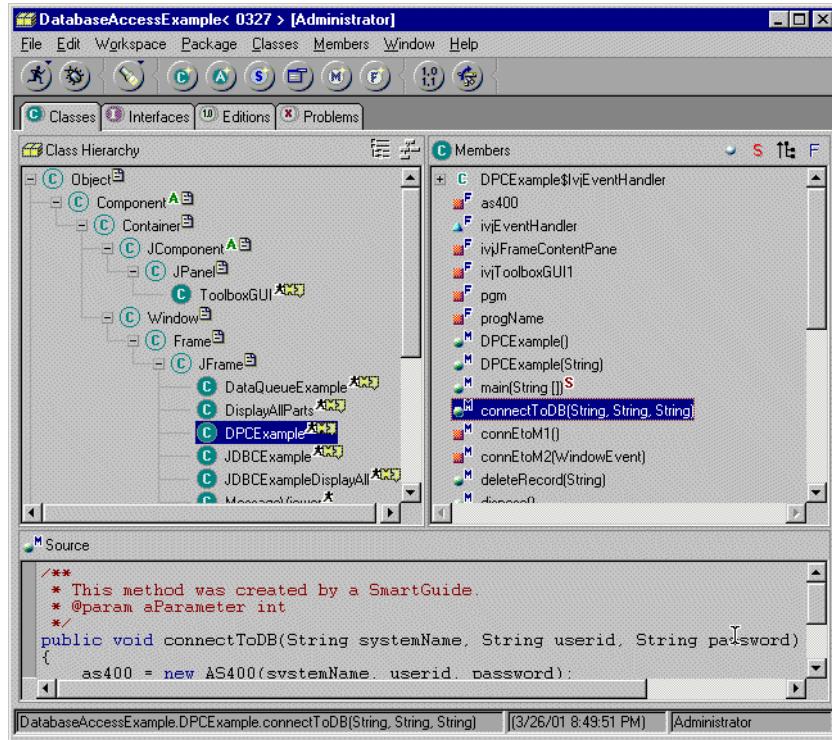


Figure 3-19 Package browser Classes view

### Type browser

The type browser is a little different in its implementation when compared with the project and package browsers. The type browser is used to display classes and interfaces (types) in the upper pane. It displays their methods, comments, and source code in the lower pane (Figure 3-20 on page 76).

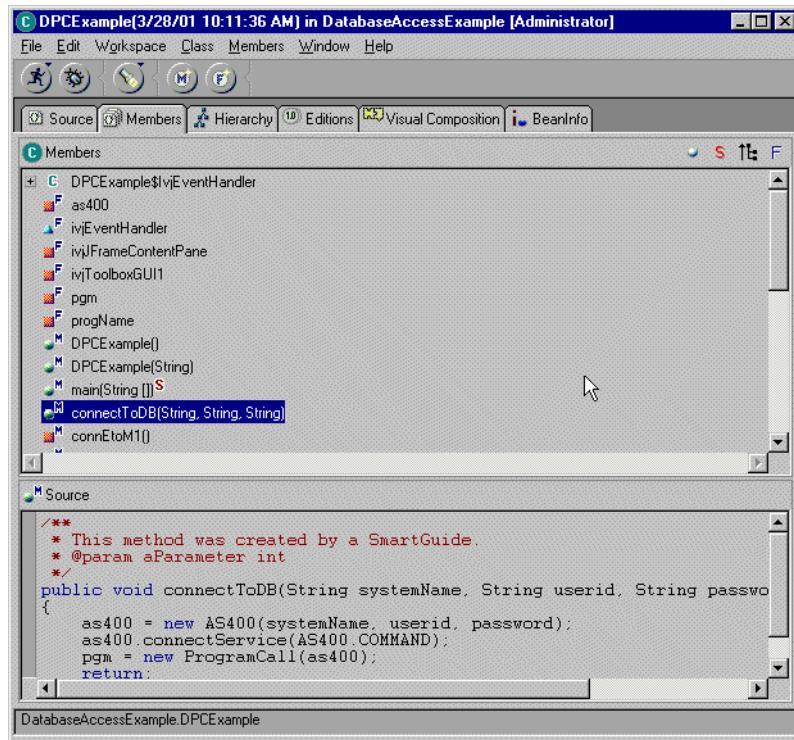


Figure 3-20 Type browser browsing class

In the Source view, you can browse and edit a complete class source file in the right pane instead of individual methods. The source view can be opened for a class or for multiple classes (for example, a package)(Figure 3-21). The hierarchy view and editions view work in the same way as the other two browsers. In the hierarchy pane, you can switch between the tree and graph layout. When displaying interfaces with the type browser, you only have four views (Figure 3-22):

- ▶ Source view
- ▶ Members Viewer
- ▶ Hierarchy view
- ▶ Edition view

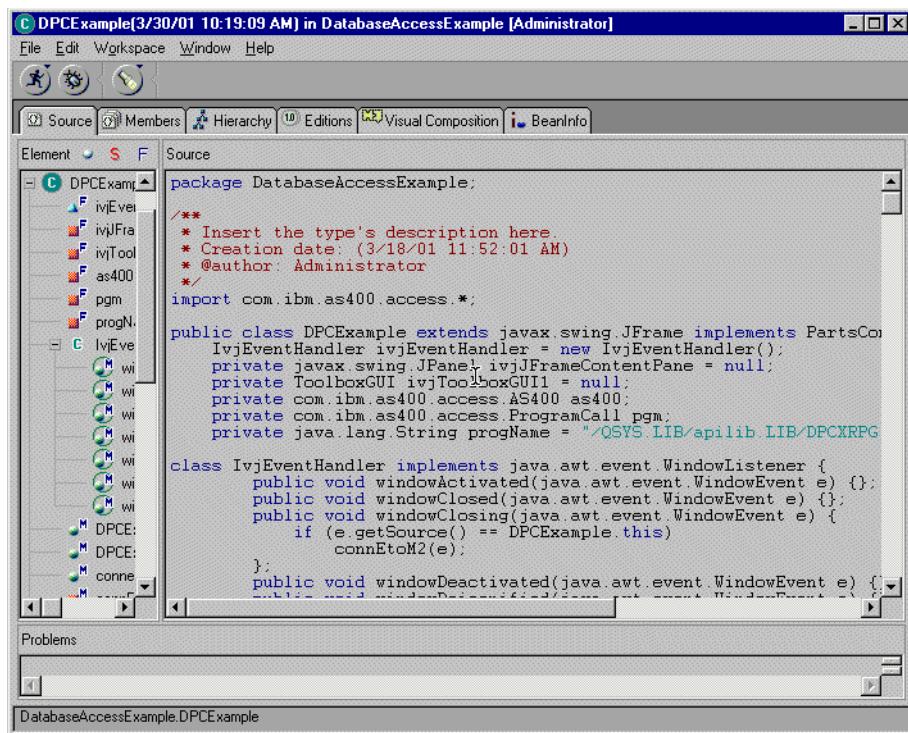


Figure 3-21 Type browser Source view

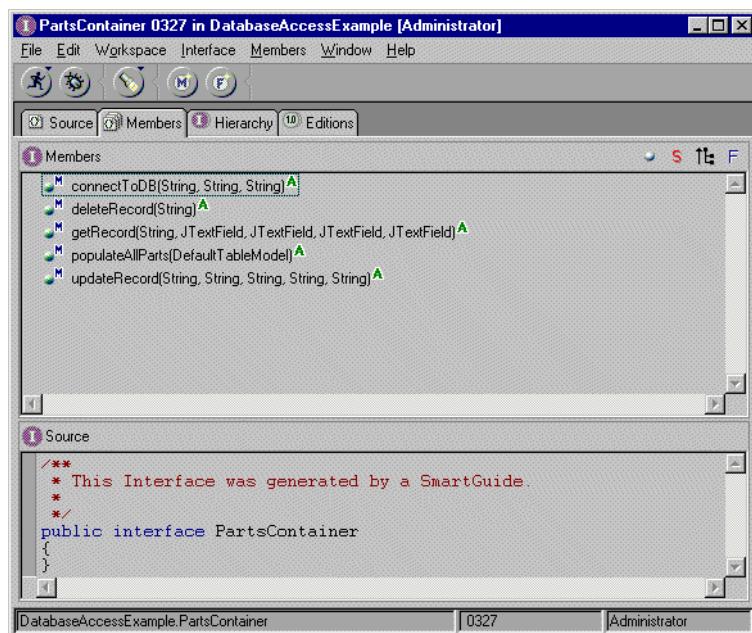


Figure 3-22 Type browser browsing interface

In addition to the browsers, there are two additional views, one for the Visual Composition Editor and one for bean information (BeanInfo):

► **Visual Composition view**

Provides a VCE for the design of classes (Figure 3-23 on page 78).

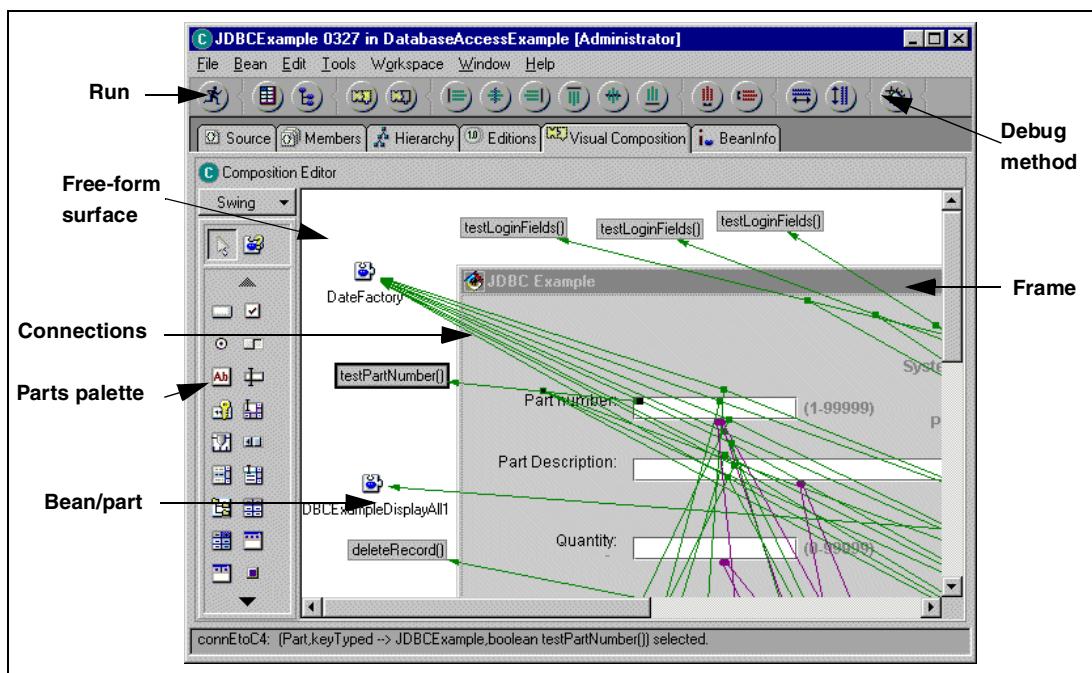


Figure 3-23 Type browser Visual composition view

#### ► BeanInfo view

Provides all information about the features that have been defined for the class (if any) and allows the BeanInfo to be modified (Figure 3-24).

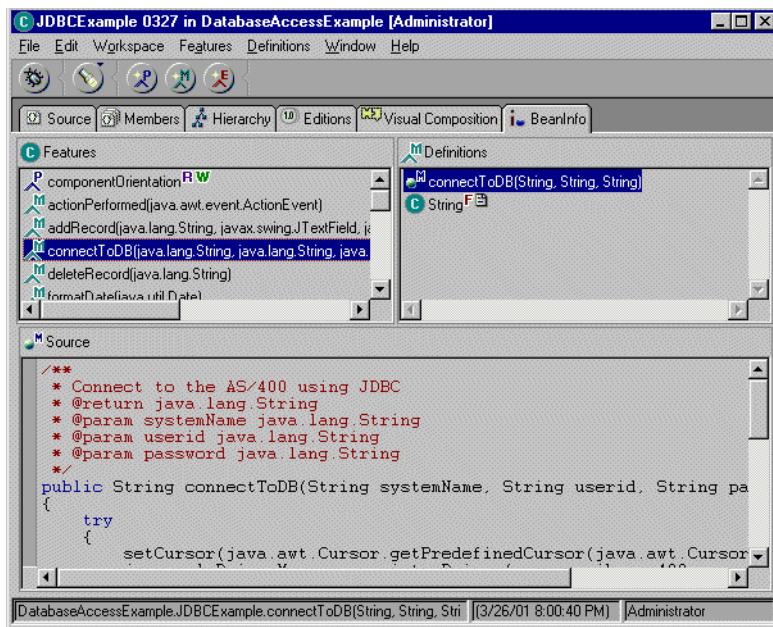


Figure 3-24 Type browser BeanInfo view

### 3.2.1 How it fits together

VisualAge for Java uses three basic components to build reusable JavaBeans and to use JavaBeans that may have been built by other tool vendors. These three components are the VCE, the Features editor, and the Script editor (Figure 3-25).

VisualAge for Java comes with a large number of reusable beans or parts that are stored either in the VisualAge image/workspace or that can be brought into the image/workspace from the repository (sometimes called the Parts/Beans Warehouse). Once a class or bean is in the image, a developer can use the VCE to connect multiple beans together to perform the required function.

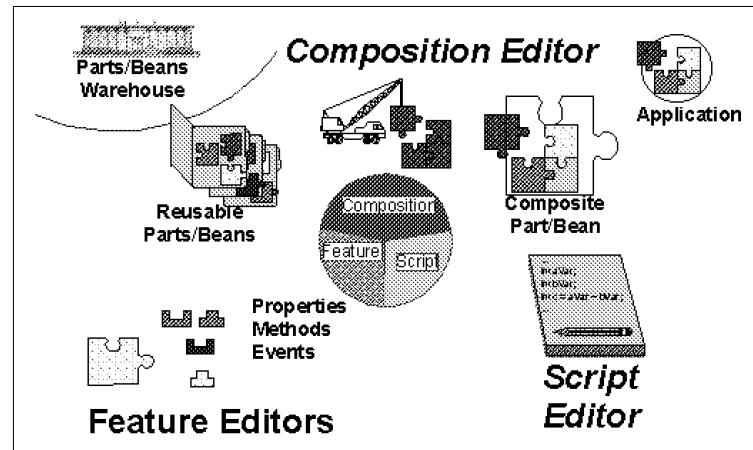


Figure 3-25 VisualAge for Java concepts

This product can also be used to develop reusable beans or modify existing beans. This is achieved by using a combination of the Feature editors for properties, methods, and events, and the Script editor for actually writing the Java code that is invoked by the various features.

### JavaBeans and classes

As discussed in Chapter 1, “Java overview and iSeries implementation” on page 1, a class is a template for objects that have similar behavior (methods) and data elements (variables, properties). To use classes in visual builders (for example, VisualAge for Java, Symantic Cafe), the class needs to have features defined for it that allow it to be connected to other beans within a visual development environment (Figure 3-26).

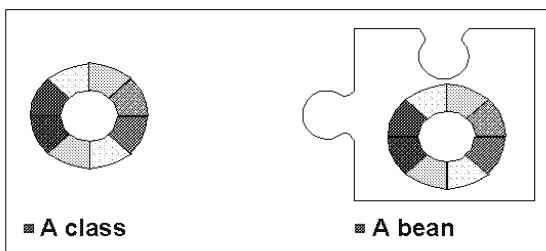


Figure 3-26 Class and bean difference

JavaBeans add standardized features and object introspection mechanisms to classes, which allow builder tools to query components (classes or groups of classes) about their properties, behavior, and events. They also allow visual builders to connect beans together that are implemented to the same JavaBeans standard. Individual JavaBeans vary in functionality, but most share certain common defining features.

These include:

- ▶ **Introspection:** Allow a builder tool to analyze how a bean works.
- ▶ **Events:** Allow beans to fire events and inform builder tools about the events they can fire and the events they can handle.
- ▶ **Properties:** Allow beans to be manipulated programmatically.
- ▶ **Methods:** Allow beans to perform functions implemented by the underlying class methods.
- ▶ **Customizing:** Allow a user to alter the appearance and behavior of a bean.
- ▶ **Persistence:** Allow beans that are customized in an application builder to have their state saved and restored.

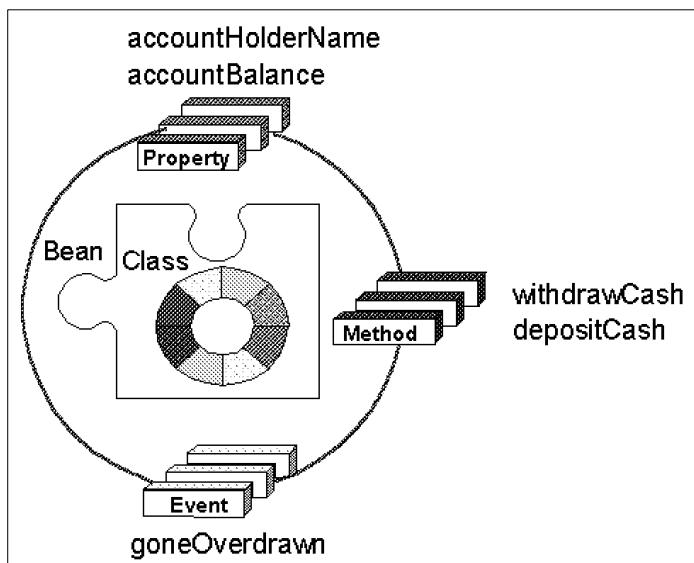


Figure 3-27 Account example

In Figure 3-27, an account class is defined with functions/methods and variables. In addition to the account class definition, bean features are defined for the following definitions:

- Variable/Property
  - AccountHolderName
  - AccountBalance
- Method
  - WithdrawCash
  - DepositCash
- Event
  - GoneOverdrawn

In this example, there is probably a one-to-one relationship between the accountHolderName and accountBalance bean properties with instance variables of the same name (defined in the class). There are also the withdrawCash and depositCash methods with bean method features of the same name. However, in addition to these four features, the bean has an event, goneOverdrawn, which is fired from within the withdrawCash method. Other JavaBeans can listen for this event before taking action. For example, an OverDrawnAccounts object may listen for account objects to fire this event. When the account object fires the goneOverdrawn event, the OverDrawnAccounts object senses this automatically (because it is listening) and takes appropriate action (sends a letter informing the account holder of the account status and charges that apply).

In Figure 3-28, there are two classes packaged as beans. The bean (for example, a push button) on the right side has a connection to the bean (for example, a list box) on the left. When the clicked event occurs, the list box performs the add function. In VisualAge for Java, this connection is made through a series of simple steps that connect the two beans together.

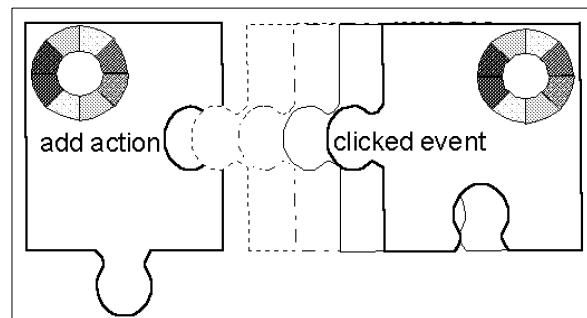


Figure 3-28 Example connection of two beans

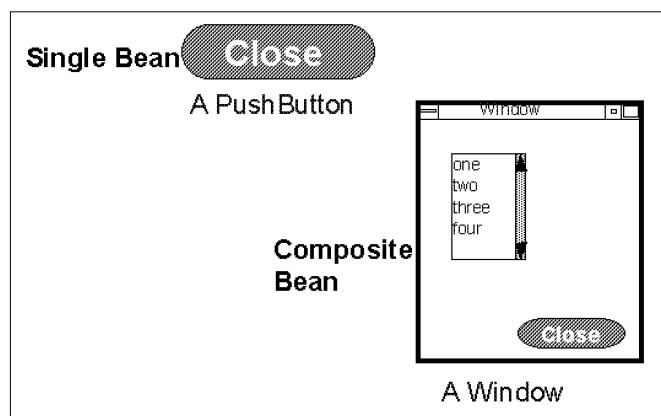
Beans can either be a single bean made up of individual beans/classes, or they can be composite beans made up of two or more classes/beans. In Figure 3-29, the push button is a single bean, where the window is a composite bean made up of a push button and a list. The same concept applies also to non-visual classes/beans. For example, an array contains a number of strings.

The previous discussion has introduced the concepts (albeit, in overview) of visual builders and of JavaBeans. In VisualAge for Java, you visually construct many modules of your application by connecting various JavaBeans using the Visual Composition Editor (VCE), VisualAge's visual builder.

There are three basic types of connections that the Visual Composition Editor provides (see Table 3-1). The return value is supplied by the connection's `normalResult` property.

*Table 3-1 Connection types*

If you want to ...	Connection type	Color	Return value
Cause one data value to change to another	Property-to-property	Dark blue	None
Call a behavior whenever an event occurs	Event-to-method	Dark green	Yes
Supply an input argument	Parameter	Purple	None



*Figure 3-29 Single and composite beans example*

A *property-to-property connection* links two property values together. This causes the value of one property to change when the value of the other changes. A connection of this type appears as a bi-directional, dark blue line with dots at either end. The solid dot indicates the target, and the hollow dot indicates the source. When the part is constructed in the running application, the target property is set to the value of the source property. These connections never take parameters.

An *event-to-method connection* calls the target method whenever the source event occurs. An event-to-method connection appears as a uni-directional, dark green arrow with the arrow head pointing to the target.

A *parameter connection* supplies a parameter value to a method by passing either a property's value or the return value from another method. This connection appears as a bi-directional, violet line with dots at either end. The solid dot indicates the target, and the hollow dot indicates the source. In addition, the parameter names are included in the connection's pop-up menu. The parameter is always the source of the connection because the parameter cannot store any values. If you connect the parameter as the target, VisualAge reverses the direction of the connection to make the parameter the source.

The Visual Composition Editor uses a dashed line to give you a visual clue so that you know when a parameter connection is needed. For example, if you connect an event to a method that requires parameter values, the connection line between the event and the method is dashed.

A connection is *directional*, meaning that it has a source and a target. The direction in which you draw the connection determines the source and target. The part on which the connection begins is the source, and the part on which it ends is the target. When you make an event connection, the Visual Composition Editor draws an arrow on the connection line between the two parts. The arrow points from the source to the target. If information can pass through the connection in both directions (as it can in property-to-property connections), a hollow circle indicates the source, and a solid circle indicates the target.

Often, it does not matter which part you choose as the source or target, but there are connections where direction is important. For example, in an event connection, the event is always the source. If you try to make an event the target, VisualAge automatically reverses it for you.

If the target of the connection takes input parameters, the connection line initially appears dashed to show that it is incomplete. Many events pass data through the connection to the target. The connection line may appear solid even if the target takes one input parameter and you have not otherwise provided one.

The target of a connection can have a return value. If it does, you can treat the return value as a no-set property of the connection and use it as the source of another connection. This return value appears in the connection menu for the connection as `normalResult`.

### 3.2.2 Building a sample application

The objective of this section is to build a simple application using VisualAge for Java. The sample application enables an end user to add parts to a list as if the user is ordering them in a parts order application.

Earlier in this chapter, VisualAge for Java was started, and you navigated past the Welcome dialog window to the Workbench window. To follow along in building the application, from the Workbench window, now perform these steps:

1. Select the **Selected** menu item.
2. Select the **Add** sub-menu item.
3. Select the **Project...** sub-menu item.

**Note:** In all future scripts, selected sub-menu items are formatted as shown in the following example:

**Selected-> Add-> Project...**

In your workbench window, this action appears as shown in Figure 3-30.

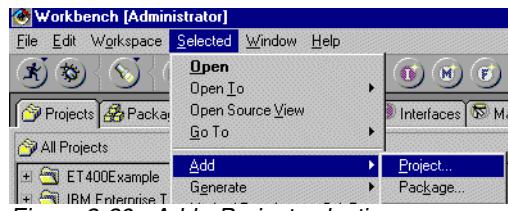


Figure 3-30 Add->Project selection

The SmartGuide Add Project window is shown (Figure 3-31). Note that you can also add existing projects from a repository.

4. Type the name of your Project, **Team01Project**, and click **Finish**.

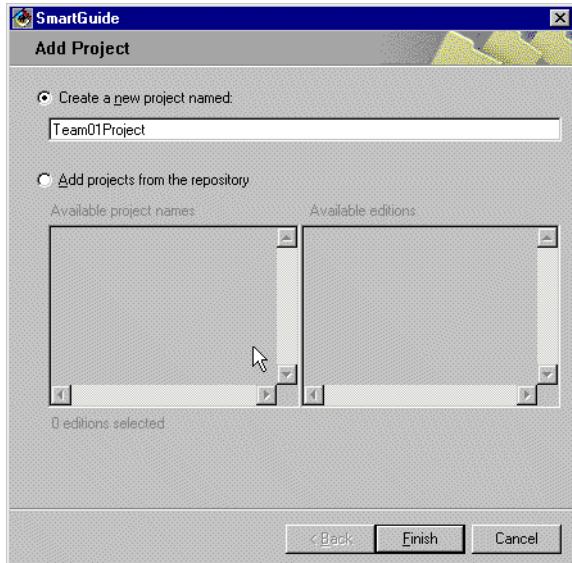


Figure 3-31 Add Project example

A project named *Team01Project* is created. You return to the Projects view of the Workbench window, and the Team01Project is highlighted.

## Opening the Team01Project

To open the project, perform these steps:

1. Bring up the Team01Project's pop-up menu (right-click).
2. Select **Open**.

The Team01Project window opens. The title of this window is "Team01Project (mm/dd/yy hh:mm:ss am) [UserID]". The time-stamp element of the window title is an indication of the date/time when this edition of the project was created. If the Team01Project window does not open up in the maximized view, maximize it now. You can see and edit more information with less mouse action if you always work with maximized windows.

## Adding a new package to the Team01Project project

To add a new package, perform the following tasks:

1. Right-click inside the **Packages** pane.
2. Select **Add-> Package...** from the pop-up menu.

The project name to which this package should be added is already filled in.

3. Enter **Team01Lab1** as the package name, and click **Finish** to create it. You can ignore the warning message about package naming conventions or allow VisualAge for Java to rename the package.

See Figure 3-32 on page 84. Note that you can also add existing packages from a repository.

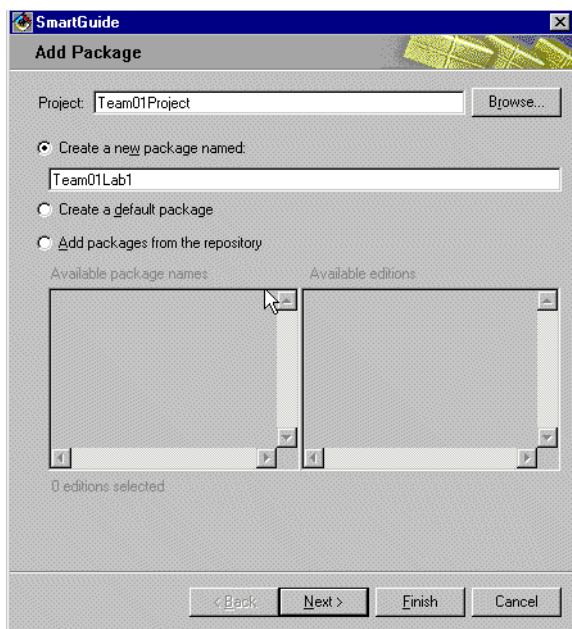


Figure 3-32 Add Package example

A new package, Team01Lab1, is created in the Team01Project. The new package is shown highlighted in the Packages pane of the Team01Project window.

### Adding a new class to a package

To add a new class, follow these steps:

1. Select the **Selected-> Add-> Class...** menu item.
2. Enter **Team01OrderEntry** as the class name.
3. Enter **javax.swing.JFrame** as the superclass name.

If you navigated as described before, the project and package name are already entered. Otherwise, you need to type them into the corresponding fields. Note that class names are case sensitive, and by convention, start with a capital letter. You are about to create a visual class, and most visual classes have javax.swing.JFrame as their super class.

To create the class, perform these steps:

1. Select the **Browse the Class when finished** check box.
2. Select the **Compose the class visually** check box.
3. Click the **Finish** button.

See Figure 3-33.

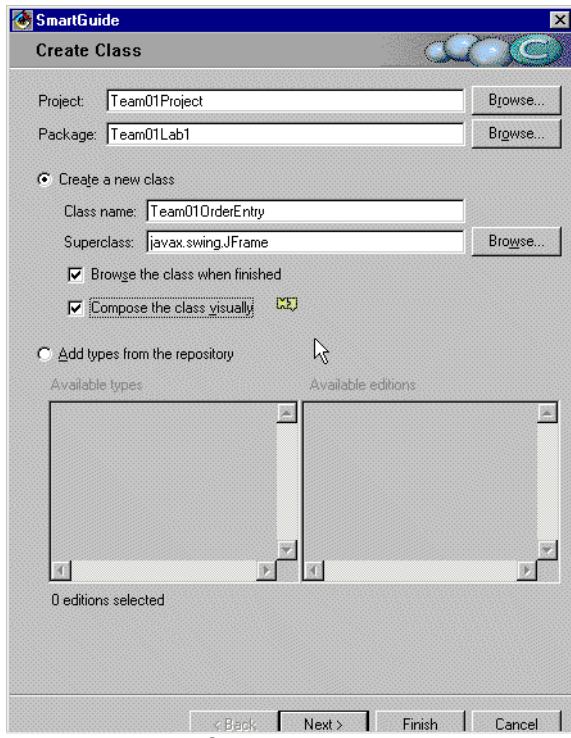


Figure 3-33 Add Class example

A new class, Team01OrderEntry, is created in the package Team01Lab1 in the project Team01Project. The new class is shown in the types pane of the Team01 Project window, and the VCE for the Team01OrderEntry class is opened and in focus.

The title of the window is “Team01OrderEntry (mm/dd/yy hh:mm:ss am) in Team01Lab1 [UserID]”. The suffix time-stamp element of the window title is an indication of the date and time when this edition of the class was created. You can also see the package to which the class belongs (Figure 3-34).

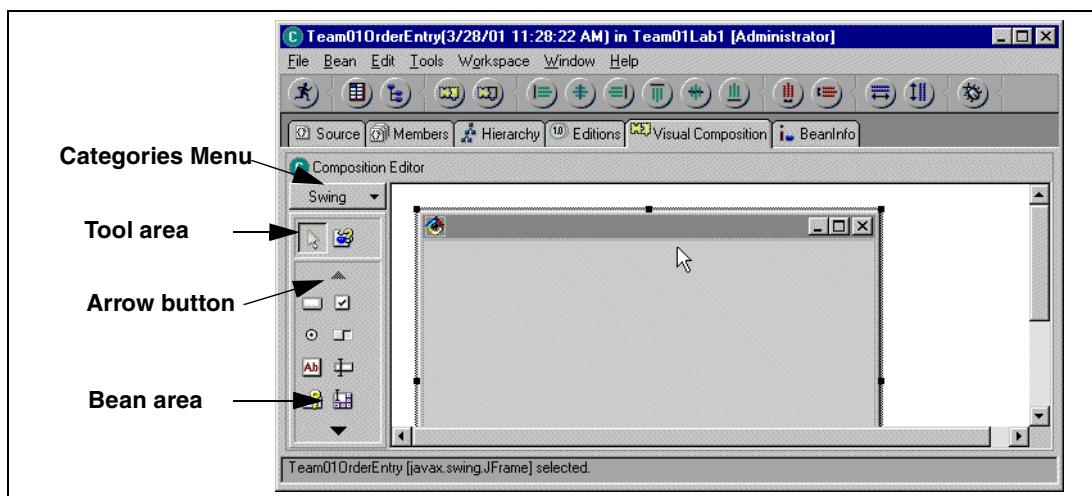


Figure 3-34 VCE example

Take a moment to review the window in Figure 3-34. There are various components in the VCE:

- ▶ **Frame:** The frame that is being built. It is usually in the top left corner of the free-form surface.
- ▶ **Free-form surface:** The white space surrounding the frame being built. The free-form surface is used to drop other parts that are not visible in the frame you are actually building (for example, a timer or another frame).
- ▶ **Parts palette:** The area on the left of the VCE window that contains:
  - *Categories menu:* Select a category of beans (for example AWT or Swing).
  - *Tool area:* Select either the selection tool or the choose bean tool.
  - *Arrow buttons:* If there is not enough space to show all beans, navigate up and down the beans of a selected group by using the arrow buttons.
  - *Bean area:* Select any bean by clicking its symbol. Now you are ready to drop it onto the design area.

## Adding the visual components to the window

The completed application for this section looks similar to the example in Figure 3-35.

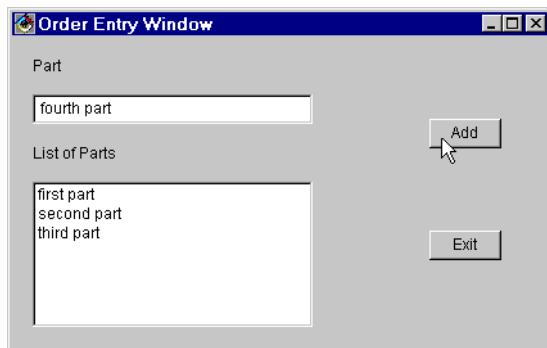


Figure 3-35 Layout example

**Note:** Do not be too concerned with the placement and alignment of parts as you are building the window. We will make it look better later.

Increase the size of the window at this point. This makes it easier to add parts. To size a part, follow this sequence:

1. Click the part to select it. There is a block in each corner, which indicates that it is selected. These are called *resize handles*.
2. Move the mouse pointer over one of these re-size handles. Click and hold the left mouse button to drag the part to its desired size.

Next, you build the graphical user interface by selecting beans from the parts palette and placing them on the window. Use the completed window shown in Figure 3-35 as a guide.

To build the interface, perform the following steps:

1. Add the following elements:
  - One text field
  - Two buttons
  - One scroll pane
  - One list

- One default list model
- Two labels

**Note:** Use the tool tip to recognize the beans in the parts palette. Move the mouse pointer over the top of the part and view the online help.

2. Make sure that the Swing category from the category selection is selected (Figure 3-36).

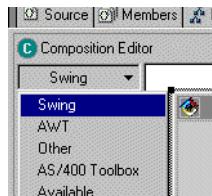


Figure 3-36 Composition Editor selecting a bean category example

To add a JTextField to the frame, follow these steps:

1. Move the cursor over the bean area, and click the **JTextField** bean.

This loads the cursor with the JTextField bean. Note the category and bean name indicated on the status line and the cursor changing into a crosshair when moved into the design area (see 1 and 2 in Figure 3-37).

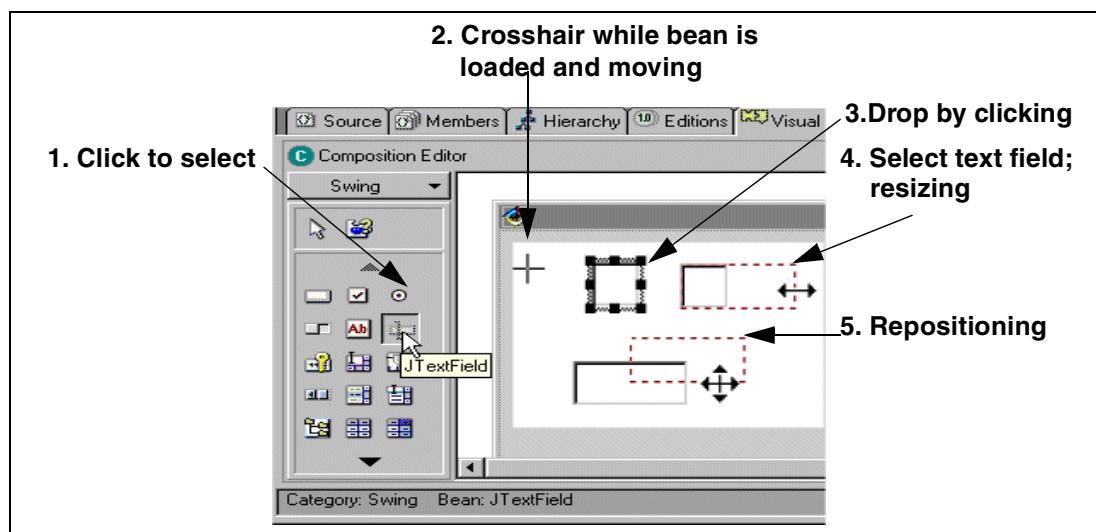


Figure 3-37 Composition Editor creating a text field example

2. Move the cursor into the design area, onto the frame, near to the upper left edge. Click to drop the JTextField into position.

The field is now drawn inside the frame and selected. You can recognize a selected object by its resize handles (see 3 in Figure 3-37). To deselect an object, simply click onto an empty spot in free-form surface around the frame.

To resize an object in the design area, follow these steps:

1. Move the cursor exactly over one of the handles.
2. Click and drag it to the desired direction.

The cursor changes into a two-headed arrow, as soon as you are in the resizing mode (see 4 in Figure 3-37).

Repositioning is similar to resizing. To reposition an object, run these steps:

1. Move the cursor exactly over one of the lines that surrounds the selected object.
2. Click on the line and drag the object to the desired place.

The cursor changes into a four-headed arrow, as soon as you are in the repositioning mode (see 5 in Figure 3-37 on page 87).

Resize the JTextField to the desired length. Follow these steps:

1. Drag the middle right resize handle to the right.
2. Reposition it by dragging the field to the desired position.

The result of these actions is a new text field on the frame (see Figure 3-37 on page 87).

To add the buttons, perform these tasks:

1. Select the **JButton** bean from the bean area.
2. Drop it onto the upper right side of the frame.
3. Drop another button just below the first one.

Because a JList doesn't directly support scrolling, we need to place the JList within a JScrollPane object, and let it deal with the scrolling. First add the scroll pane by following these steps:

1. Select the **JScrollPane** bean from the bean area.
2. Drop it onto the lower left side of the frame under the text field.
3. Resize the list by dragging the lower right handle to the lower right corner.

To add the list, follow these steps:

1. Select the **JList** bean.
2. Move and stop over the scroll pane, drop the JList bean onto the scroll pane object.

Next, we add a default list model to hold data that will be displayed in the list. To add the default list model, follow these steps:

1. Click the **Choose Bean** icon, and then select the **javax.swing.DefaultListModel** class.
2. Drop it onto the *free-form* surface.

To add the label, complete these steps:

1. Select the **JLabel** bean.
2. Drop it just above the text field you created earlier.
3. Drop another Label bean above the list.

Your application should now look similar to the example in Figure 3-38.

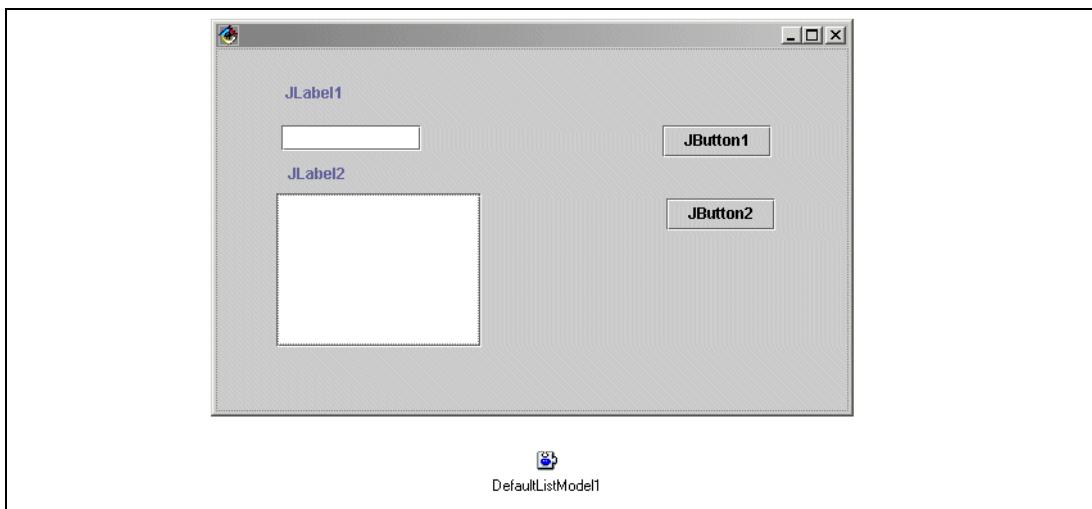


Figure 3-38 Using the VCE to build an application

**Note:** If you want to create an object multiple times inside the design area (two buttons and two labels in our example), there are other, more efficient ways to do this. For example, pressing the Control key during the selection of a bean from the bean area causes the cursor not to be unloaded when the bean is dropped. This way you can add several beans of the same type while only selecting it once. To unload the cursor, click the selection tool, or directly select another bean from the bean area. Another way to create multiple beans of the same type is to select an object of the same type from the design area by clicking it once. Now you can copy the selected object using one of these three different ways:

- ▶ From the menu bar, select **Edit->Copy** and then **Edit->Paste** (slow). Now your cursor is loaded with the same bean that you selected before, and you are ready to drop it onto the design area.
- ▶ Use the shortcut keys indicated near the menu items from the Edit menu (fast). In our example, press **Ctrl+C** and **Ctrl+V**.
- ▶ Press and hold down the **Ctrl** key, and *drag* the selected *object* to a new position (very fast).

Use the Delete key for a shortcut



Figure 3-39 Composition Editor shortcut keys

Becoming familiar with shortcut keys improves your speed in visual programming greatly. The most frequently used actions all have shortcut keys (Figure 3-39).

### Making the frame look good

This section shows you how to improve the appearance of the frame. First, we work with the labels. Move the cursor over the **JLabel1** just above the **JTextField1** and double-click.

This action opens the properties window where all of the properties (features) of the bean can be set. There is a drop-down menu under the title bar, where the name of the selected bean appears. In the properties window, there are two columns. The left one indicates the property name, and the right one displays its current value.

**Note:** Whenever you cannot find a property, make sure that the **Show expert features** check box is selected (see Figure 3-40 on page 90).

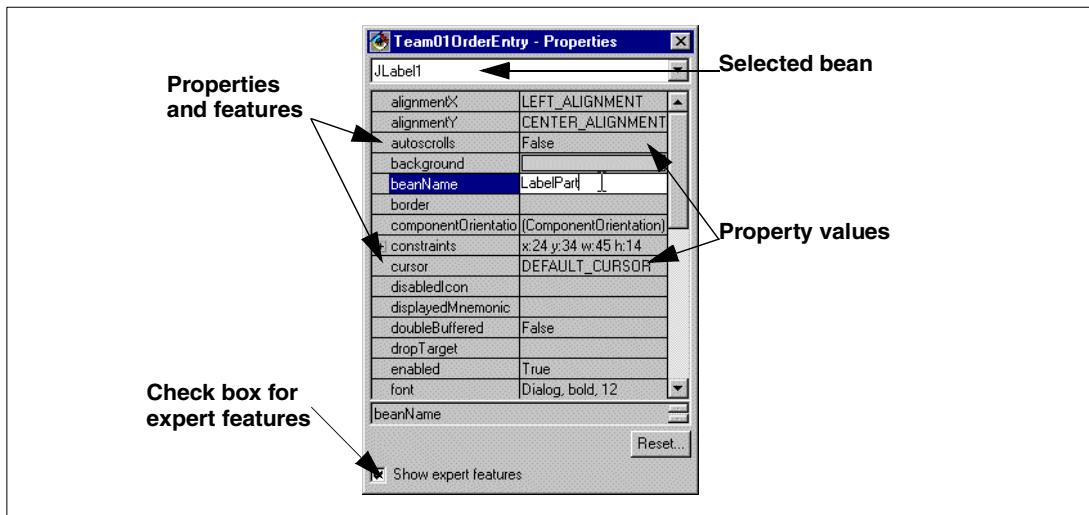


Figure 3-40 Composition Editor Properties window example

Now we are going to change two property values for bean JLabel1 in the properties window. Change the text JLabel1 to Part and the bean name JLabel1 to LabelPart.

**Note:** It is helpful to change a bean's name. It is much easier to search for a LabelPart than for a JLabel9 out of 15 labels inside the Java source code, which is generated later. Giving the label a bean name that refers to the corresponding object makes it also easier to determine which label belongs to which object. This applies to any bean you build in the VCE, as well as for buttons, text fields, frames, and so on.

To change the JLabel1 property value, perform these steps:

1. Click the property value for the text property.
2. Overwrite JLabel1 with Part.
3. Click the property value for beanName.
4. Overwrite JLabel1 with LabelPart.
5. Click the X on the upper right corner to close the property window.

When changing the text property, you see a small button on the right end of the property value. If you click it, it brings up the text property editor, which can be used to externalize strings. We do not need this option in our examples. Simply overwriting the text works for us. Note that for different property types, you have different property value editors. For example, consider a color property value editor, a boolean property value editor, or a font property value editor (see Figure 3-41).

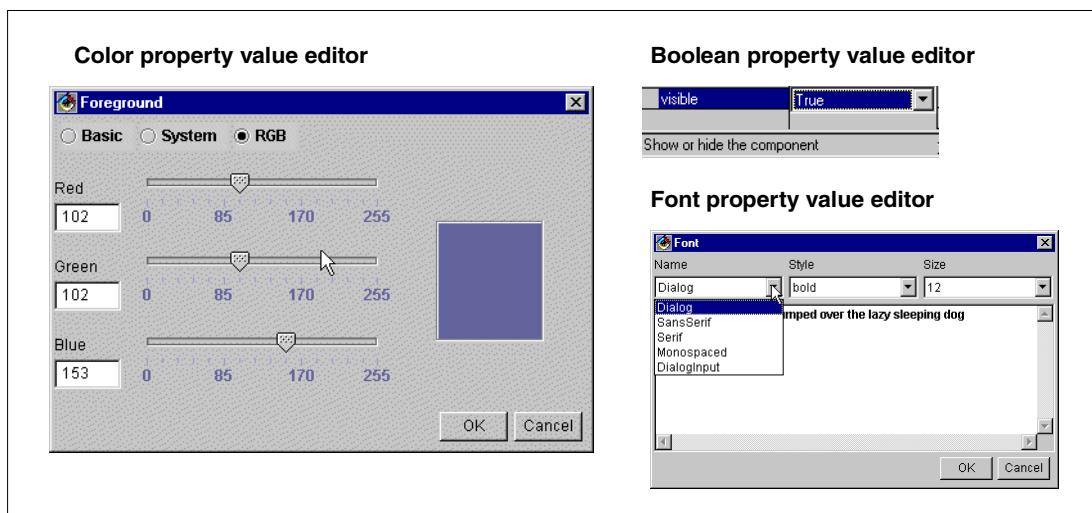


Figure 3-41 Composition Editor property editor examples

The label we changed before should now appear as Part and have the name LabelPart. Check this by selecting (clicking) the bean again, and see which text appears in the status line. Following the same procedure, you can now change all the other beans in the design area. Keep the property window open while you select one bean after the other. The changed values are saved, even if the property window is not closed. To change the background color, use the color property value editor as shown in Figure 3-41.

To complete property changes, perform these steps:

1. Open the property window for the Team01OrderEntry (double-click the title bar of the frame).
2. Change its title to **Order Entry Window**.
3. Change its background to **lightGray**.
4. Select **JTextField1**, and change its beanName to **Part**.
5. Change its background to **White**.
6. Select **JLabel2**, and change its beanName to **LabelListParts**.
7. Change its text to **List of Parts**.
8. Select **JScrollPane1**, and change its beanName to **ScrollPaneParts**.
9. Select **JList1**, and change its beanName to **ListParts**.
10. Change its background to **White**.
11. Select **JButton1**, and change its beanName to **ButtonAdd**.
12. Change its label to **Add**.
13. Select **JButton2**, and change its beanName to **ButtonExit**.
14. Change its label to **Exit**.

After this exercise, your frame should look similar to the example in Figure 3-35 on page 86. It is a good idea to save your work now. To save your work, perform either of the following options:

- From the menu bar, choose **Bean->Save Bean**.
- Press **Ctrl+S**.

Now an information window appears that illustrates the saving process. VisualAge for Java is saving the designed frame and generating the Java source code for it. Switch to the Methods view of the type browser, and look at the generated class and methods. You can display the source code of a method by clicking the methods name in the upper pane. The source code is displayed in the lower pane.

Switch back to the VCE view since there is some resizing, repositioning, and alignment work left to do. Look at the tool bar before you complete the frame (Figure 3-42).



Figure 3-42 Composition Editor tool bar

You can learn the function of each smarticon by moving the cursor slowly over it and viewing the help text inside the tool tip. It shows you which action will be taken when the smarticon is clicked. Depending on which object or how many objects are selected, the smarticons are enabled or disabled. For now, you need to use the green alignment, red distribution, and blue match tools.

You can select multiple objects when sizing, positioning, or aligning the components. To select multiple objects, hold down the Ctrl key and click sequentially all of the desired objects. If you are working with multiple object selections, the last selected object is always the reference object for all of the other objects in your selection. The size and position of the reference object affects the size and position of all the other objects in your selection. This applies when you are resizing or repositioning the reference object or when you are using one of the alignment, distribution, or match tools. You recognize the reference object by its filled (black) resize handles, while the other objects in the selection have hollow (white) handles (Figure 3-43).

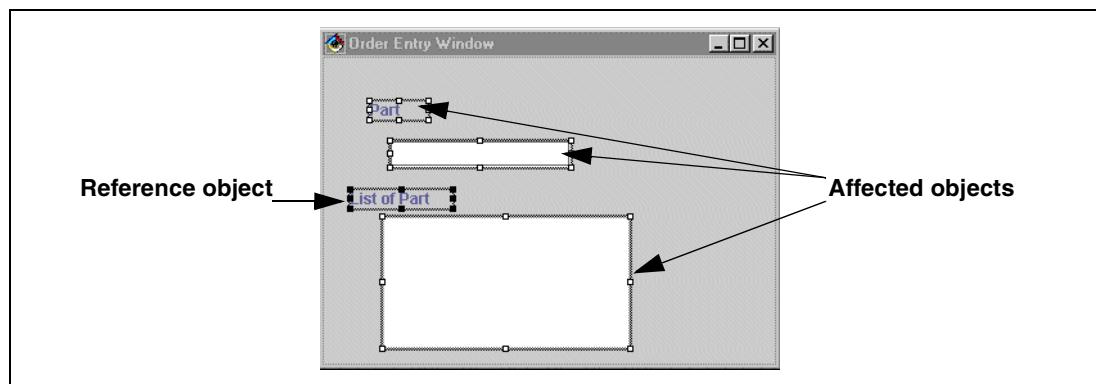


Figure 3-43 Composition Editor multiple bean selection example

To size the Part and ScrollPaneParts beans, complete these steps:

1. Select, resize, and reposition the **ScrollPaneParts** bean until you are satisfied with the width and position of it.
2. Hold down the Ctrl key, and click the beans in this order: **Part** and **ScrollPaneParts**.
3. Click the **Match Width** smarticon.

The result of this action is that the width of the Part bean matches the width of the ScrollPaneParts bean.

To align the beans, perform these steps:

1. Hold down the Ctrl key, and click the beans in this order: **LabelPart**, **Part**, **LabelListParts**, and **ScrollPaneParts**.
2. Click the **Align Left** and the **Distribute Vertically** smarticons.

Observe the beans that are aligned to the left with the ScrollPaneParts bean and equally distributed in vertical direction inside the frame.

To resize the ListParts label, follow this sequence:

1. Select and resize the **LabelListParts** bean so that its entire text can be seen.
2. Deselect the LabelListParts bean by clicking on the white space around the frame.

To align the buttons, complete these steps:

1. Hold down the Ctrl key and click the beans in this order: **Button Add** and **Button Exit**.
2. Click the **Align Left** and **Distribute Vertically** smarticons.

To run the application, perform the following tasks:

1. Click the **Run** smarticon. The application is saved, generated, and started. The developed frame appears.
2. Close the window to return to the VCE.

You now know all you need about designing a GUI with the VCE. You can experiment with the techniques described in this chapter to become familiar with the VCE. Using your style and GUI building skills, design the window according to your own preferences. After this, you are ready to test the application. It does not have any functionality built in yet, but you can see how it appears.

**Note:** Because ListParts overlaps ScrollPaneParts, sometime it's difficult to select ScrollPaneParts. In this case, you can use the **Beans List** smarticon, and select the ScrollPaneParts from the list in the pop-up window (Figure 3-44). Then the ScrollPaneParts can be selected in the frame.

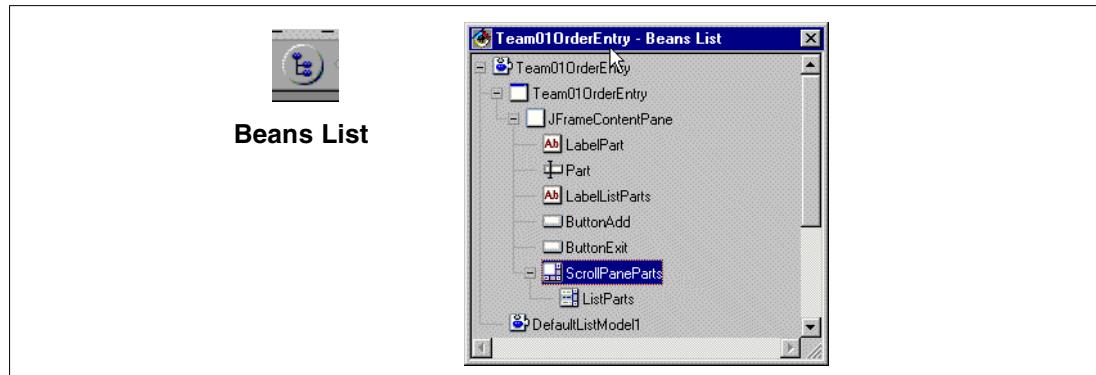


Figure 3-44 Beans List example

### Adding the function

First, you need to make the Exit button work. If a user clicks the Exit button, the frame should close in the same way as if the X button in the top right corner of the frame was clicked. The only visible connection at this time provides exactly that function. To find out what a connection stands for, left-click the line connecting the two objects together and view the text displayed in the status line (Figure 3-45 on page 94). Select the connection from the frame bean to the `dispose()` method.

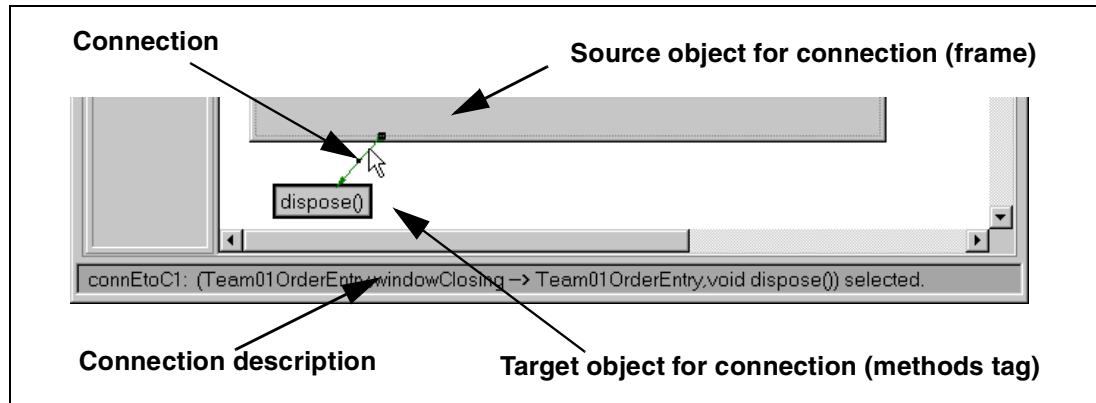


Figure 3-45 Composition Editor Connection example

You need to build the same kind of connection to make the Exit button work. Whenever you want to connect an object, select it and right-click to display its pop-up menu. Selecting Connect brings up the features available to you as defined in the bean. They are displayed in the connection source pop-up window.

For the Exit button, find the actionPerformed feature, which listens or watches for the default action being performed for the part. For a button, the default action is the button that is clicked. After selecting the connection source, a dashed line with the *spider* on its end is shown. The spider allows you to connect objects (beans) together by moving over the design area and clicking to select a target object for the connection. The target object to which you can connect is marked by a dashed box.

Select the frame as the target in this example. As soon as you select that object, the connection target pop-up window is displayed. From this window, choose the desired feature of the target object. In this case, you want the frame to be closed or disposed, so select the `dispose()` feature. As a result of this, a green connection is displayed between the Exit button and the frame (Figure 3-46).

To connect the Exit button to the Frame dispose method, perform these steps:

1. Move the cursor over the Exit button and click to select it.
2. With the cursor still over the Exit button, right-click to bring up the button's pop-up menu.
3. Select **Connect->actionPerformed**.
4. Move the spider to the title bar of the frame and click.
5. From the target feature window, select **dispose()**.
6. Click **OK**.
7. Run the application and test the function of the Exit button.

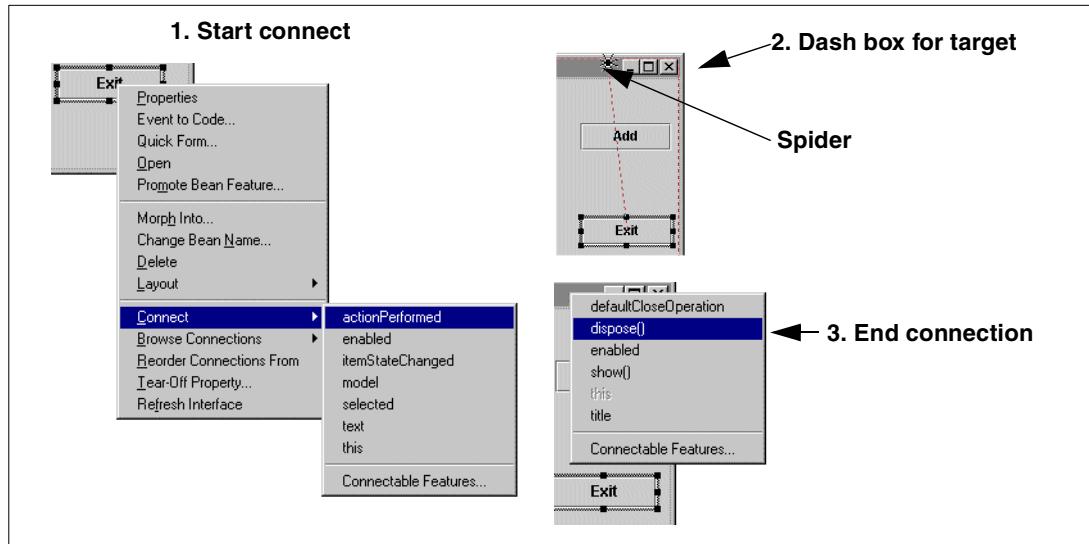


Figure 3-46 Connecting example

To explain all of the possible ways of creating a connection, we delete and rebuild the connection two more times. These changes do not have any effect on the function of the Exit button. Yet it still works in the same way.

To re-create the Exit connection, complete these tasks:

1. Click the connection from the Exit button to the frame.
2. Press the Delete key to delete the connection.
3. Select the **Exit button** and right-click to bring up its pop-up menu.
4. Select **Connect->actionPerformed**.
5. Move the spider to an empty spot in the free-form surface, and click it.
6. From the target feature pop-up window, select **Connectable Features...** and choose **dispose()**. See Figure 3-47.
7. Click **OK**.

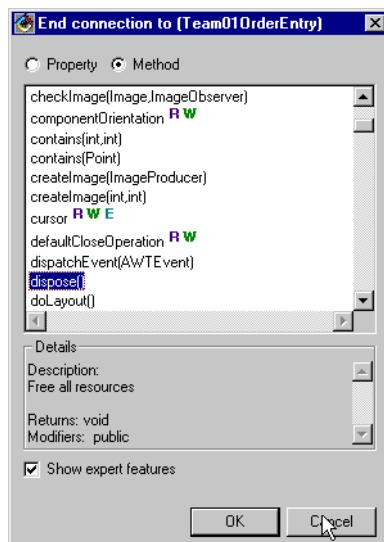


Figure 3-47 Connectable features example

This connection points to the border of the design area. Compare the text shown in the status line for this connection with the previous one. Notice that there is no difference between them. Select the Connectable Features whenever you want to investigate a bean in the VCE. This works for both the starting and ending features of a connection.

Note, that depending on the type of bean selected, you can choose between the properties, methods, and events that you want to connect. If you cannot find a feature, make sure that the *Show expert features* check box is checked. You can also change an existing connection by double-clicking on it. This action brings up the connection properties window. There you can change source and target features and set parameters for the connection if required. Note that this window also has a check box for expert features (see Figure 3-48).

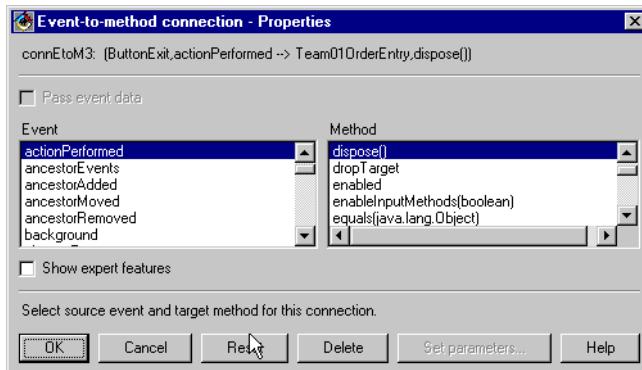


Figure 3-48 Connection property editor example

The third way to create a connection is to connect a source feature directly to the method source code. This can be achieved by selecting Event to Code... from the connection target pop-up window. That brings up a source editor from which you can select the source event, the target method, and the target method class. For the frame that you are designing, you can also create a new method for the target feature in the connection. The source editor also appears when you double-click a method tag (grey box with methods name) inside the design area. To see all the details of the source editor, refer to Figure 3-49.

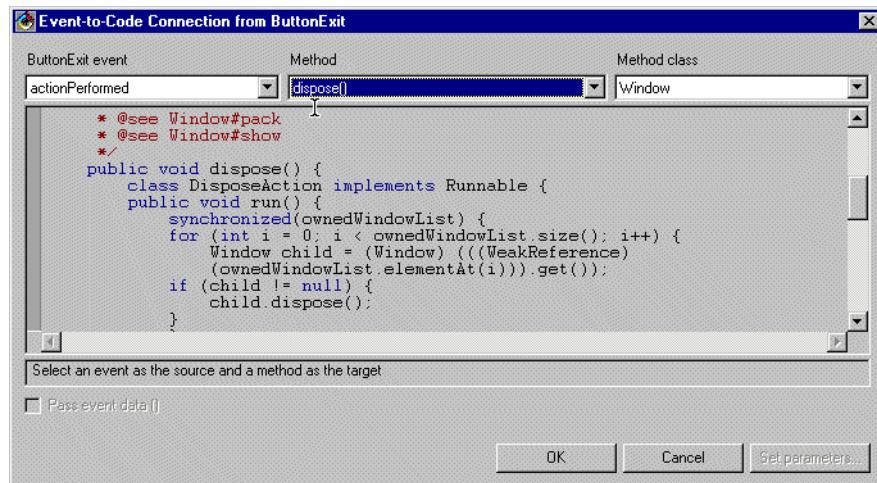


Figure 3-49 Connection source editor example

To create the connection again, perform the following steps:

1. Click the connection from the Exit button to the design area.
2. Press the Delete key to delete the connection.

3. Select the **Exit button**, and right-click to bring up its pop-up menu.
4. Select **Connect->actionPerformed**.
5. Move the spider to an empty spot of the free-form surface, and click it.
6. From the target feature pop-up window, select **Event to Code...**, which brings up a source editor.
7. From the Method class drop-down list, select **Window**.
8. From the Method drop-down list, select **dispose()**.
9. Click **OK**.
10. Press **Ctrl+S** to save your work.

Note that the connection also shows its resize handles when selected. You can change the source and target of a connection by selecting one of the resize handles at the end (a spider is shown) and moving it onto another object. With the resize handles in the middle (four-headed arrow is shown), you can drag the line of the connection to any desired spot in the design area. This way, you can improve the readability of your visual design in case you have many connections to draw. Try to create parallel lines, and avoid crossing them as much as possible, for less a less confusing, more clear design.

Since our frame does not have its own `dispose()` method, we have to choose the `dispose` method from the super class, which is `javax.swing.JFrame`. In this source editor, you can also display, edit, and create methods for your bean. The result of the previous actions is a connection that looks the same as the first one that was generated by the VCE. The only difference is its starting point, the Exit button.

Now we need to relate the `ListParts` object to the `defaultListModel1` object, which holds data for the `ListParts` object. To do so, perform the following steps:

1. Move the cursor over the **ListParts** object and click to select it.
2. With cursor still over the **ListParts** object, right-click to display the pop-up menu.
3. Select **Connect -> model**. Move the spider to the `defaultListModel1` object and click.
4. Select **this** from the pop-up menu, and left-click.

You are now ready to visually perform the function to add text entries from the `TextField` to the list. To do so, complete the following steps:

1. Move the cursor over the **Add** button and click to select it.
2. With the cursor still over the Add button, right-click to bring up the Buttons pop-up menu.
3. Select **Connect->actionPerformed**.
4. Move the spider to `defaultListModel1` and click.

The connection target pop-up window appears.

What happens when the Add button is clicked? You want to add the text/string to the list that was entered in the `TextField`. Select **Connectable Features->addElement(java.lang.Object)** from the pop-up window and click. A dashed green connection is displayed between the Add button and the `defaultListModel1` object.

You have now completed half of this connection. You told VisualAge that when the Add button is clicked, a string should be added to the list. However, you did not specify which string to add. To complete the connection, follow these steps:

1. Move the cursor over the dashed green connection between the Add button to the `DefaultListModel` object.
2. Click over the connection to select it. Selection handles are shown along the connection to show that it has been selected.

3. With the cursor still over the connection (but not on a selection handle), right-click to bring up the Connections pop-up menu.
4. Select **Connect->obj**.
5. Move the spider over **Part**.
6. Click and select the text.

A purple arrow joins Part to the green connection. Do not forget that VisualAge applies color to each connection depending on its type.

You have now completed the window for this section. Test it out by clicking the **Run** smarticon. Enter some values in the Part, and check if the Add button adds them to the ListParts JList. Also try using the Exit button. Figure 3-50 shows the completed VisualAge frame (your window should look similar).

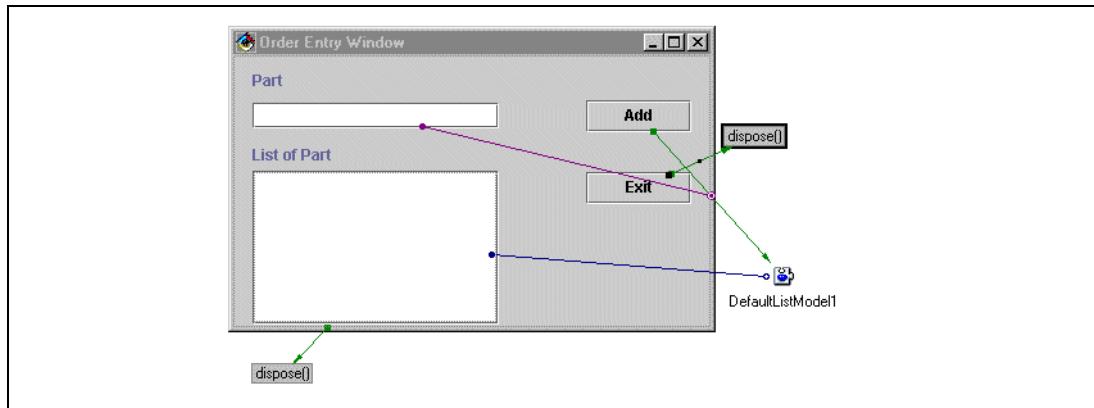


Figure 3-50 VCE example

### Creating a version for your application

Until now, you have been working inside an open edition of the project, package, and class. When your example proves to work in the tests, you may want to make a version out of it. By doing this, you can extend our application inside a new edition and return to the last version whenever you need.

To version the application, perform these steps:

1. Click the **Hierarchy** tab.

The class hierarchy is displayed showing the Team01OrderEntry class and its super classes.

2. Click the **Show Edition Names** smarticon.

The class Team01OrderEntry is followed by a time stamp, which indicates that this is an open edition of the class.

3. Click **Team01Lab1.Team01OrderEntry** to select it.

4. From the menu, select **Classes->Manage->Version....**

5. Make sure the **Automatic** radio button is selected, and click **OK** (Figure 3-51).

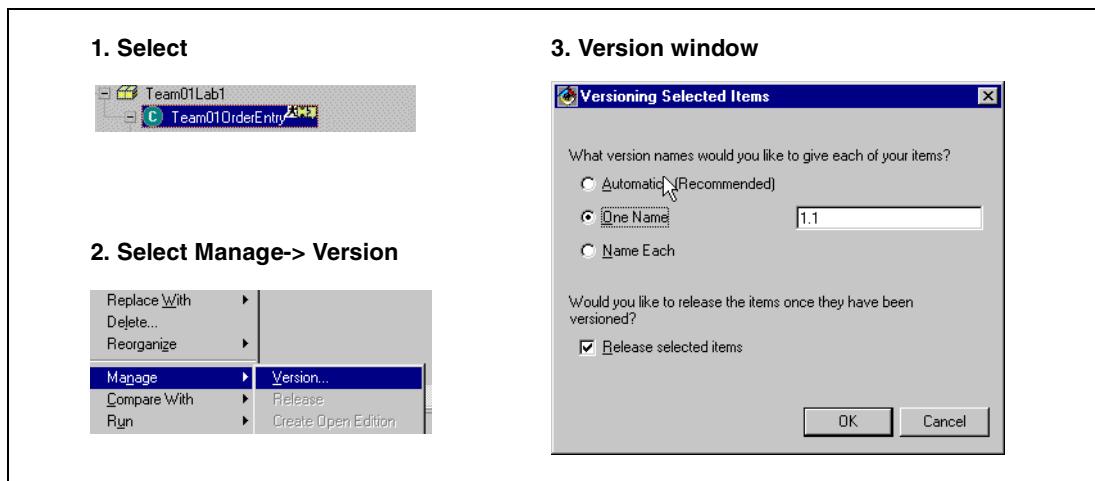


Figure 3-51 Create version example

Your class now has a version. Instead of the time stamp, the class name is followed by its version number. To develop the class further, you need to create another open edition of it in one of two ways:

- ▶ Select the class and from the menu bar: **Classes->Manage->Create Open Edition**.
- ▶ Wait to be automatically prompted for that action the next time you save the class with any new changes.

To create an open edition, complete these tasks:

1. Click **Team01Lab1.Team01OrderEntry** to select it.
2. From the menu bar, select **Classes->Manage->Create Open Edition**.

The new open edition of Team01OrderEntry is created. You are ready for the next enhancements. To go back to any version of the class later, select the class from the same screen. Then, select **Classes->Replace With->Another Edition....**. Choose the desired edition from the window containing all of the available editions.

## Extending the application

Next, we extend the application by performing these actions:

1. Add a quantity field.
2. Modify the behavior of the Add button so it invokes a script to concatenate the part and quantity details, and displays them in the list.
3. Add a Delete button to delete existing entries from the list.
4. Enable or disable the Delete button when an item is selected or deselected in the list.
5. Add a Java script breakpoint, and modify the code when the breakpoint is invoked.

The completed development window should appear similar to the window shown in Figure 3-52 on page 100.

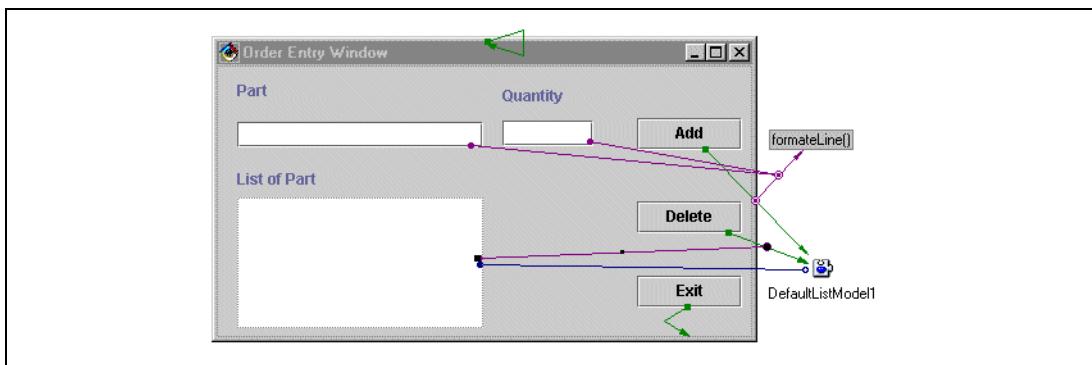


Figure 3-52 VCE finished example

Click the **Visual Composition** tab to go back to the VCE view. Complete this series of steps:

1. Add a Delete button to the window:
  - a. Select the **JButton** bean from the bean area and drop it between the ButtonAdd and the ButtonExit.
  - b. Double-click the button to bring up its properties window.
  - c. Change the beanName property to **ButtonDelete**.
  - d. Change the label property to **Delete**.
  - e. Change the enabled property to **False**.
  - f. Allow the properties window to remain open.
2. Add a text field that allows the quantity of parts to be input:
  - a. Select the **JTextField** bean from the bean area and drop it level with, and a little to the right, of the Part text field.
  - b. Resize and reposition this field until you are satisfied with the result.
  - c. In the properties window, change the beanName property to **Quantity** and the background property to **white**.

**Note:** Do not close the Properties window.
3. Add a label and change its text property to **Quantity**:
  - a. Select the **JLabel** bean from the bean area, and drop it above the quantity text field.
  - b. In the properties window, change the beanName property to **LabelQuantity** and the text property to **Quantity**.
  - c. Left align the LabelQuantity with Quantity, and middle align it with Part.
  - d. Close the Properties window.

We add items from the text fields (Part and Quantity) to the ListParts JList by using a script. Therefore, the current connection from the ButtonAdd (actionPerformed) to the defaultListModel1(AddElement(java.lang.Object)) is no longer needed. To delete it, follow this sequence:

1. Move the cursor to the green connection from ButtonAdd to defaultListModel1.
2. Click the connection to select it.
3. As soon as you see the resize handles of the connection appear, press the Delete key.
4. Click **Yes** when prompted by the confirmation message.

The connection and any connections it supported are deleted.

## Writing a new Java method

To add both the part and quantity text to the list, we write a method to concatenate the two text fields together. For this, you must switch to the Methods view.

To create a new method, perform these tasks:

1. On the tool bar, click the **Create Method or Constructor** smarticon.
2. Select the **Create a new method** radio button.
3. In the Method Name entry field of the Method Properties window, type:  
`String formatLine (String aPart, String aQuantity)`
4. Click the **Finish** button.

See Figure 3-53.

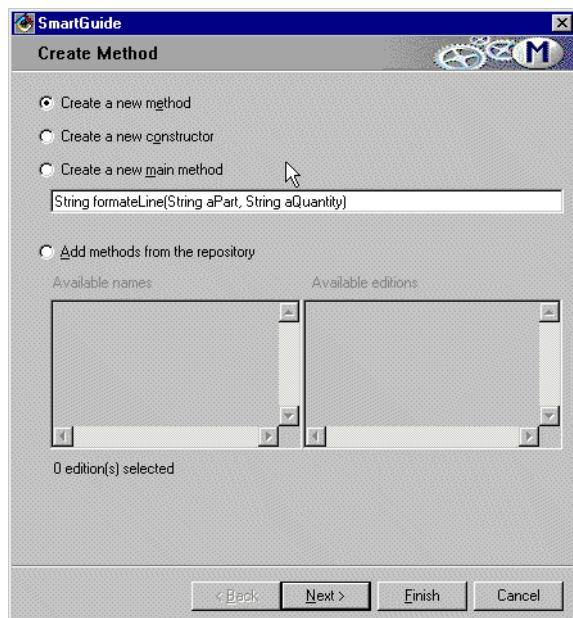


Figure 3-53 Create Method example

A new method called `formatLine` is created that takes two string parameters (`aPart` and `aQuantity`) and returns a string (the concatenated string). At the moment, it returns null, but you want it to return the concatenated string. Replace the null-value by an expression that concatenates the string `aPart` with a string containing a single blank character and the string `aQuantity`.

To update the `formatLine` method, complete these steps:

1. Inside the source pane, double-click the word **null**.
2. Type: `aPart + " " + aQuantity;`
3. Press **Ctrl+S** to save the method.

The method's source should appear as shown in Example 3-1 on page 102.

---

*Example 3-1 The formatLine method*

---

```
/**  
 * Insert the method's description here.  
 * Creation date: (3/28/01 3:05:02 PM)  
 * @return java.lang.String  
 * @param aPart java.lang.String  
 * @param aQuantity java.lang.String  
 */  
String formatLine(String aPart, String aQuantity)  
{  
    return aPart + " " + aQuantity;  
}
```

---

Make sure that your source code matches this example. Click the **VCE** notebook tab to return to editing the frame.

In the next step, you rebuild the connection for the function to be executed when the Add button is clicked. Complete these steps:

1. Select the **ButtonAdd** and right-click.
2. From the popup menu, select **Connect->actionPerformed**.
3. Move the cursor over defaultListModel1 and click.
4. Select **Connectable features...** from the pop-up menu.
5. select **addElement(java.lang.Object)**.

A green dashed connection is shown from the ButtonAdd to defaultListModel1. A dashed connection means that the connection requires parameters that have not yet been supplied. For this reason, you have to set the **obj** object of the connection by using these steps:

1. Select the dashed connection from ButtonAdd to defaultListModel1, and right-click.
2. From the pop-up menu, select **Connect ->obj** and move the spider to the free-form surface. Right-click and select **Parameter from Code**.
3. In the source editor, select **formatLine(String, String)** from the **Methods** drop-down list.
4. Click **OK**.

A purple dashed connection is shown from the middle of the last connection to **formatLine(String, String)**, which means that this connection requires parameters that have not yet been supplied. For this reason, you have to connect the text properties of Part and Quantity to the connection by using these steps:

1. Select the dashed connection and right-click it.
2. From the pop-up menu, select **Connect->aPart**. Move the spider over Part JTextField.
3. Click and select the text.
4. Select the connection again and right-click it.
5. From the pop-up menu, select **Connect->aQuantity**, and move the spider over Quantity JTextField.
6. Click and select the text.
7. Click the **run smarticon** to test the application.

You should be able to add parts to the list using the script that you just created. Return back to the VCE.

In the next step, you build the connection for the function to be executed, when the Delete button is clicked. Complete these tasks:

1. Click the **Delete** button to select it, and right-click to bring up its pop-up menu.
2. Select **Connect->actionPerformed**.
3. Move the spider over **defaultListModel1**.
4. Click and select **removeElement(java.lang.Object)**.

There is a new dashed green connection from the Delete button to the defaultListModel1 object.

To supply the parameter for the connection, perform these steps:

1. Click the **ListParts** tJTextField to select it.
2. Right-click and select **selectedValue** from the pop-up menu.
3. Move the spider over the middle of the connection from the **ButtonDelete** object and **defaultListModel1 object**. A small dashed box indicates that there is a possible end-point for the connection.
4. As soon as the small dashed box appears in the middle of the connection, click and select the **obj** feature.

The dashed green line should change to a solid green line.

Now, change the behavior of the Delete button. It should be enabled as soon as a selection from the ListParts has been made. Perform the following steps:

1. Click the **ListParts** JList to select it.
2. Right-click and select **Connect->Connectable Features...** to bring up the source feature selection window.
3. Select the **valueChanged** event. This event is fired when the selected item changes.
4. Move the spider over the **ButtonDelete** object. Click and select the enabled property. Now there is a dashed green connection between the ListParts object and the ButtonDelete object. You need to set a boolean parameter for this connection to be completed.
5. Double-click the dashed green connection to open the connection properties editor window.
6. Click the **Set parameters** button to open the constant parameter value window.
7. Set the value to “True”, and click **OK** twice.

This action enables the button when an item is selected from the ListParts JList object. You also want to disable the ButtonDelete whenever a part is deleted from the ListParts object. To do this, add another connection from and to the ButtonDelete object using these steps:

1. Select the **ButtonDelete** object, and right-click. Select **Connect->actionPerformed**.
2. Move the spider over the **ButtonDelete** object, and click. Select **enabled**.
3. Double-click the dashed green connection.
4. Click the **Set parameters** button.
5. Set the value to “False” and click **OK** twice.

Now, test the new function by adding some part and quantity items to the list. Try to select a few items from the list to see if you can delete them. The Delete button should only be enabled when an item is selected in the list. Keep the test window running, and continue with the next section.

## **Debugging, setting breakpoints, and changes ‘on the fly’**

This section discusses the *VisualAge for Java Debugger* and some of the useful features it offers when analyzing and debugging your applications.

To change the code on the fly, complete these steps:

1. Return to the VCE.
2. Click the **Members** notebook tab to switch to the methods view.
3. Click the **formatLine(String, String)** method to select it.
4. In the source pane, change the line that returns the concatenated string to:  

```
return aPart + " : " + aQuantity;
```
5. Press Ctrl+S to save the method.
6. Select the test window, and add another part and quantity item to the list.

The code you changed was used to add this new part. Your test window should look similar to the one shown in Figure 3-54.

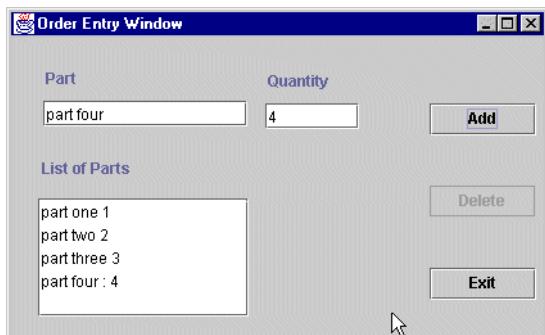


Figure 3-54 Finished application example

Note that the last added part or quantity has an additional blank and colon in between the part and quantity, as defined in the **formatLine(String, String)** method.

To set a breakpoint, follow this sequence:

1. Return to the Members view of the type browser.
2. Edit the **formatLine(String, String)** method.
3. Move the cursor to the line that returns the concatenated string.
4. Right-click and select **Breakpoint**.
5. Click **OK** on the dialog box that appears.

A blue breakpoint marker is shown to the left of the line. This is the point where the code stops prior to running it and opens up a debugger window. If the blue breakpoint marker does not appear, you probably were not in the first column or you were on an incorrect line (see Figure 3-55).

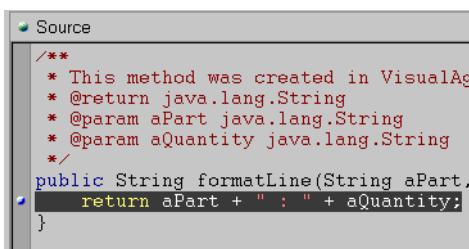


Figure 3-55 Breakpoint example

Add another part or quantity item to the running test window.

The debugger window appears. The code stops prior to executing the statement. In the three upper panes, the left-hand pane (All Programs/Threads) shows a tree view of all current threads when the debugger was invoked. The threads are expanded with their call stacks with the most recent method at the top. The center pane (Visible Variables) shows the variables that are accessible, and the right pane (Value) shows the value of the currently selected variable. The bottom pane (Source) shows the current line in the source code of the current method (Figure 3-56).

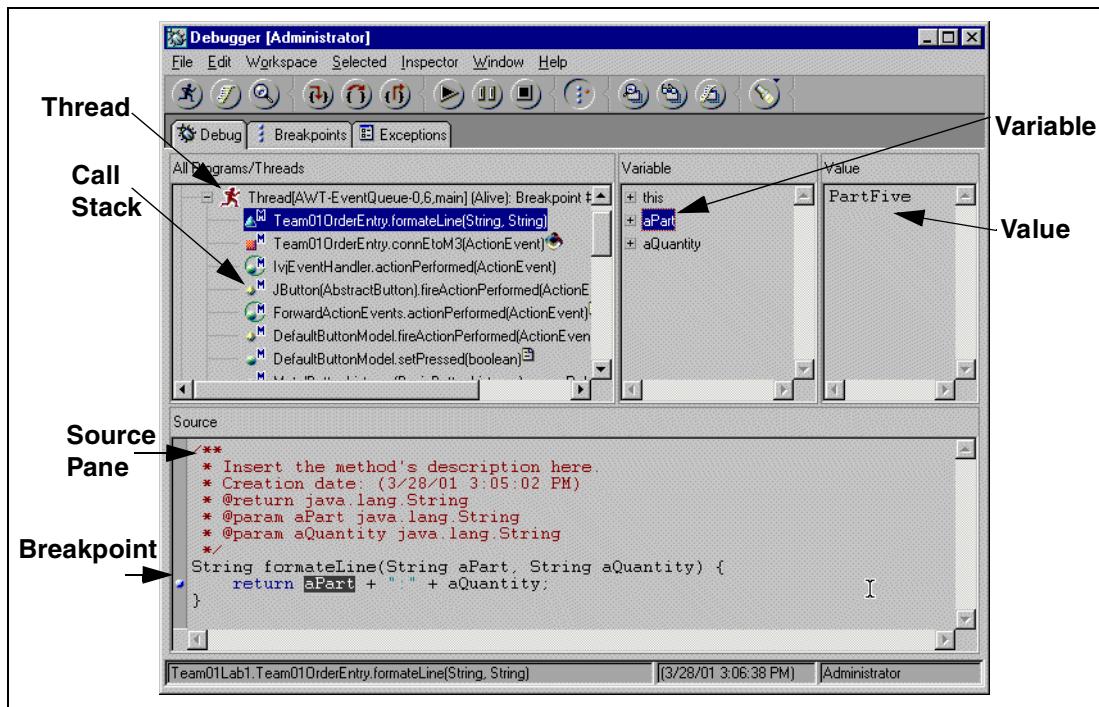


Figure 3-56 VisualAge for Java Debugger

To view a variable, click the **java.lang.String aPart** variable in the visible Variables pane.

The Value pane is updated and displays the string value of whatever you typed into your Part. As you are aware, a string is an array of characters.

Click the plus (+) sign to expand the Part variable you selected. Then, expand the resulting char[] value entry. Select entry 0, then 1, then 2, and so on.

When you examine the single characters inside the array, you may recognize that all of the characters from your typed part appear one after the other in the Value pane (Figure 3-57).

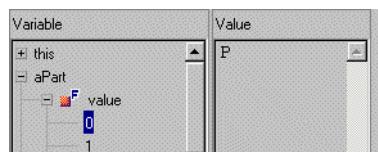


Figure 3-57 Examining the variables example

Click the next entry in the All Programs/Threads pane under the Team01Lab1.Team01OrderEntry.formatLine(String, String) entry.

This should be one of the connEtoCx(ActionEvent) types. Notice that the Visible Variables, Value, and Source pane are all updated. In the Source pane, the actual code that called the current method is highlighted (Figure 3-58 on page 106).

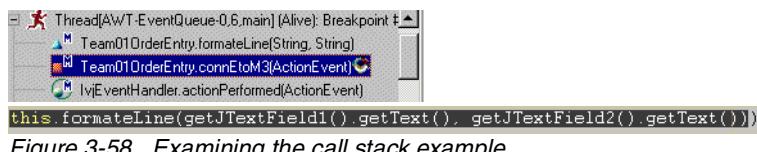


Figure 3-58 Examining the call stack example

You can also find the method that you are currently looking at in the Methods view of the Team01OrderEntry type browser.

To change the code, perform these steps:

1. Reselect the top entry in the All Programs/Thread pane (not the thread above), **Team01Lab1.Team01OrderEntry.formatLine(String, String)**.
2. Modify the code so that the string " : " now reads " :- ". Save the method.
3. Click the **Breakpoints** notebook tab to switch to the Breakpoints view.
4. From the menu, select **Methods->Clear** to remove the breakpoint.
5. Switch back to the **Debug** view, and click the **Resume** smarticon in the tool bar of the debugger window. The debugger window blanks out since that thread has now run to completion.
6. Close the debugger window, and navigate back to the running test window.

Your part or quantity entry is added, with the " :- " separator between the part and quantity. This example shows that anywhere you have a method source window, you can modify the method, save it, and run it immediately with the updated code.

To test the Delete button, complete these steps:

1. Select an entry in the ListParts of the running test window.
2. Click the **Delete** button.

The Delete button should be enabled as soon as an entry from the ListParts is selected. The entry should be deleted, and the Delete button should again be disabled until you select another entry in the ListParts.

### Closing the application and versioning

To version the application, follow this sequence:

1. Close the **Team01Lab1.Team01OrderEntry** type browser.
2. Switch to the **Projects** view of the workbench window.
3. Select the class expanding your project and package until Team01OrderEntry is seen.
4. Select **Selected->Manage->Version...** from the menu bar.
5. Version the class. Either accept the default version name or enter your own.
6. Select **File->Save Workspace** to save the workspace.

You should now have a working knowledge of the VisualAge for Java Integrated Development Environment. This will help you follow the programming examples discussed in this Redbook.

### 3.2.3 Team development

In the team development environment, giving a number of developers access to the same code requires control mechanisms to endure consistency, correctness, and currency of the code. Without some control mechanisms, the entire project will rapidly descend into chaos.

VisualAge for Java provides two different solutions to the problem:

- ▶ A shared repository on a central server accessible by all developers. Change control works at the object level and is based on object ownership. This function is only available in Enterprise Edition.
- ▶ An interface to third-party tools for storing, managing, and controlling files or data. It is this interface that has changed in the VisualAge for Java Version 3.5. Previously it was known as *Software Configuration Management (SCM)*. It is now called *External Version Control*. It is also available in both the Enterprise Professional Editions.

This Redbook introduces the first solution.

## Shared repository on a server

Figure 3-59 shows the components that make up a shared repository solution.

In the shared repository environment, all source code is stored in a shared repository on a central server. The central repository server runs team server, EMSRV, which manages concurrent access to the shared repository on the server. In version 3.5, the team server is supported on Windows 2000 and Red Hat Linux, in addition to Windows NT, AIX, OS/2, Netware, HP-UX, and Solaris.

Team members connect from the Workbench on their client machine to the shared repository.

Once connected, they can perform the following tasks:

- ▶ Find program elements in the shared repository, including those developed or owned by other team members
- ▶ Bring various editions of those program elements into their own workspace
- ▶ Create, change, test, and version program elements
- ▶ Release editions of program elements that they own for other team members to use as a common baseline
- ▶ Selectively replace editions of program elements in their workspace with other editions released by other members of the team

For an individual, team development allows a developer the freedom to develop code independently from the rest of the development team, yet remain within the scope of the overall project. A developer can recall at any time a history of individual changes made to any component made within the developer's image/workspace, plus the ability to retrieve prior versions of a component should this be appropriate. This total flexibility in development allows a developer to try things out in the knowledge that at any time, a prior frozen version of a component can be recalled. The component to be recalled can be an individual method, an entire class/interface, a package, or a complete project.

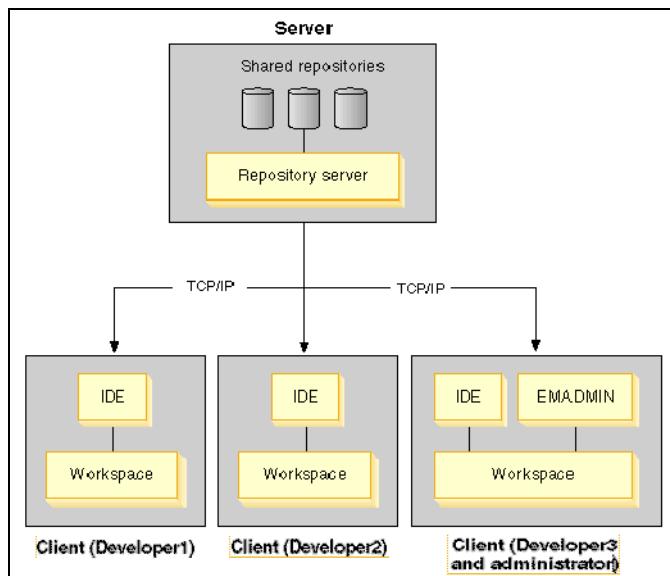


Figure 3-59 Example of a shared repository

Version control within the team development provides the facilities to freeze the development of a component (class, package, or project) so that no changes can be made to that component. This is extremely useful when setting checkpoints for components within a development cycle.

With the Enterprise Edition, multiple developers can, if appropriate, work on any component (project, package, class, or method) concurrently. In a normal check-in, check-out philosophy, this is impossible. But, within the VisualAge for Java Enterprise Edition, this can be achieved. Despite this flexibility, component integrity is never compromised. For further information, see the VisualAge for Java documentation.

In the Professional Edition, each developer has a unique repository that stores every component available, although the developer may only have a subset of components in the image. However, in the Enterprise Edition, every developer can share a common repository that allows all of the work to be shared and accessed concurrently, online, and in real time.

Just as in the Professional Edition, the Enterprise Edition records all changes made to any component and who made that change. In the Enterprise Edition, there are facilities to enable the access control rights for individual developers to every component within the repository. Therefore, because of the ease of development with fallback facilities, the development in a Rapid Application Development (RAD) type environment is positively encouraged by the tool, with all the management controls should they be necessary.

The configuration of VisualAge for Java places a development image/workspace on the client and a repository on the client/file server in the Professional Edition. In the Enterprise Edition, a shared repository has to be placed on a shared server. The repository holds a copy of every version of every component for the development team, where the image/workspace contains only the requested version of a sub-set of components. For example, Developer1 may work on GUI projects, packages, and classes, where Developer2 may be work on iSeries access projects, packages, and classes. The shared repository (the Enterprise Edition) holds every edition and version of all these components. For example, Developer2 image/workspace holds only the iSeries access components, not the GUI components.

In a team development environment (Figure 3-60), changes made by a developer to any component are written back immediately to the repository. Therefore, the component change is immediately made available to all other developers who may be using the component. On a nightly basis as part of the regular systems management procedures, the repository should be backed up to external media.

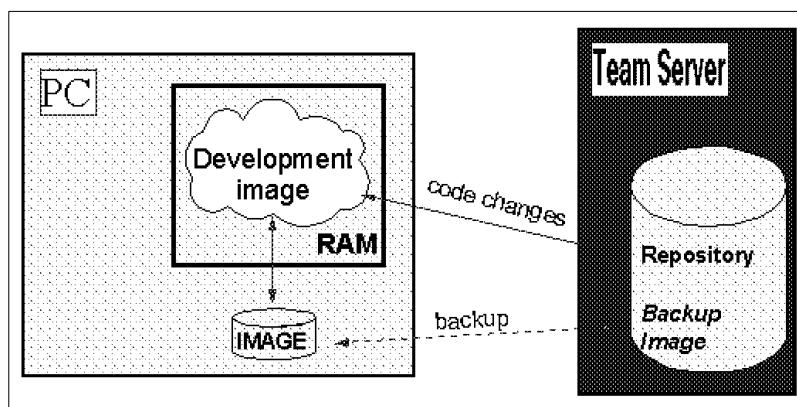


Figure 3-60 Team development backup procedures

When a developer starts VisualAge for Java, the image/workspace is copied from disk into memory. The developer works with this copy of the image when adding, deleting, or changing components. It is vital that the developer saves this

“in-memory” image to disk on a regular basis (for example, once per hour). It is not catastrophic if the developer has a system crash after an entire series of changes because every component is still available in the repository. However, rebuilding the image from scratch may be time consuming.

In addition, at regular intervals (for example, at lunch time and at the end of the day), each individual developer should copy their working image/workspace to the iSeries server. These, again, should be backed up on a nightly basis.

The team development facilities enable editing and creating versions of components. This is a simple process where the developer can create a version of a component at any time where a version is a frozen component that cannot be changed.

Figure 3-61 shows three separate versions of the component. The developer can assign each version a unique name. In the example, the versions are 1.0, 1.1, and 2.0. As with most things in VisualAge for Java, a component can be any class/interface, package, or project. The developer explicitly versions these components. Methods are the exception because every change to them that is saved causes the creation of a new edition of the method.

The big question is: If a component is a version and a version is just another name for a frozen component that cannot be changed, how do you change a component? The answer to this is to create a new edition of the component. An edition of a component is editable, but the original version of the component remains in the repository should the developer need to go back to it at any time. The process for creating, freezing, and changing a component (for example, Class A) is described here:

1. Create Class A (it is created as an edition):
  - a. Write methods.
  - b. Define variables.
2. Version Class A as Class A 1.0:  
Class A is frozen and cannot be changed.
3. Edition Class A:  
Class A can now be edited again, but version 1.0 is still available if it needs to be restored.
4. Version Class A as Class A 2.0

See Figure 3-62 on page 110.

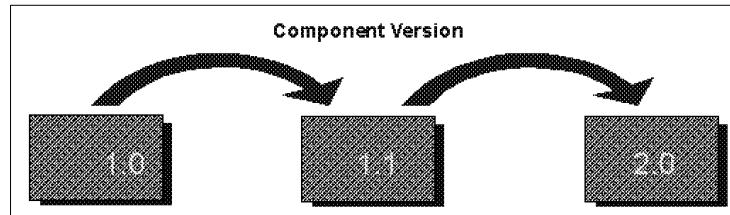


Figure 3-61 Team development versions

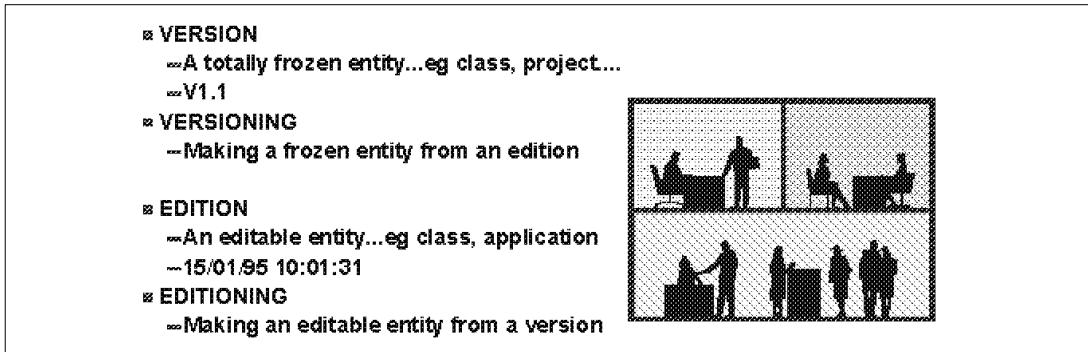


Figure 3-62 Team development process

For more detailed information on team development related issues, such as External Version Control, please refer to the following sources:

- ▶ Select **Components ->Team Development ->Concepts ->Team development** in the VisualAge for Java Help
- ▶ *VisualAge for Java Enterprise Version 2 Team Support, SG24-5245*

### 3.2.4 Applets and Applet Viewer

The VisualAge for Java Applet Viewer is incorporated into the IDE. This enables a developer to develop Java applets and to test them without bringing up a separate Web browser (for example, Netscape). The Applet Viewer is a primitive viewer and should only be used for debugging purposes with the final testing being performed in a real-life Web browser. However, because the Applet Viewer comes with VisualAge for Java, it supports the level of the JDK supported by the IDE (currently JDK 1.2.2). You may not be certain of this level of support in some Web browsers.

VisualAge for Java has an applet creation SmartGuide that is accessed through its Create Applet smarticon on the tool bar (Figure 3-63). The applet creation SmartGuide walks the developer through the process of creating an applet and completing the tasks that usually are hand-coded into the applet.

One of the windows that is displayed as part of the SmartGuide is included here as an example of the type of information the applet creation SmartGuide can process. The SmartGuide — Applet Properties window allows the setting of applet/application and thread details. Many applets can be run as applets and stand-alone applications. In the latter case, a `main()` method needs to be created. In addition, should the applet perform a long running task or repeatable task (such as repeating animation), we advise that you write this as a separate thread. Again, the SmartGuide provides the option of creating the applet to use its own thread.

To create an applet, perform these steps:

1. Select the **Team01Lab1** package from project **Team01Project** in the workbench window.
2. Click the **Create Applet** smarticon on the tool bar.
3. Enter **SampleApplet** for the Applet name, and click the **Finish** button.
4. In the VCE, add a label in the design area.
5. Change the text property of the label to **Hello World**, and adjust its width.
6. Save the Applet and switch to the **Hierarchy** view of the type browser.

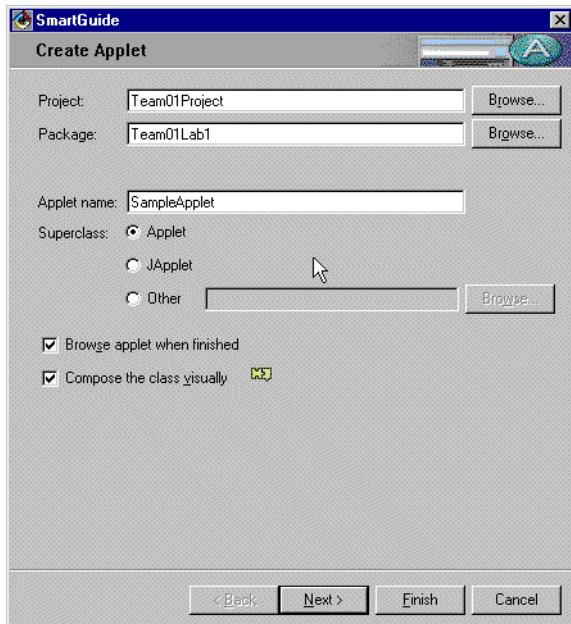


Figure 3-63 SmartGuide: Create Applet

After the applet is saved, you can see its place in the class hierarchy in the type browsers Hierarchy view. As you expect, the applet inherits its required methods from the java.applet.Applet class (Figure 3-64).

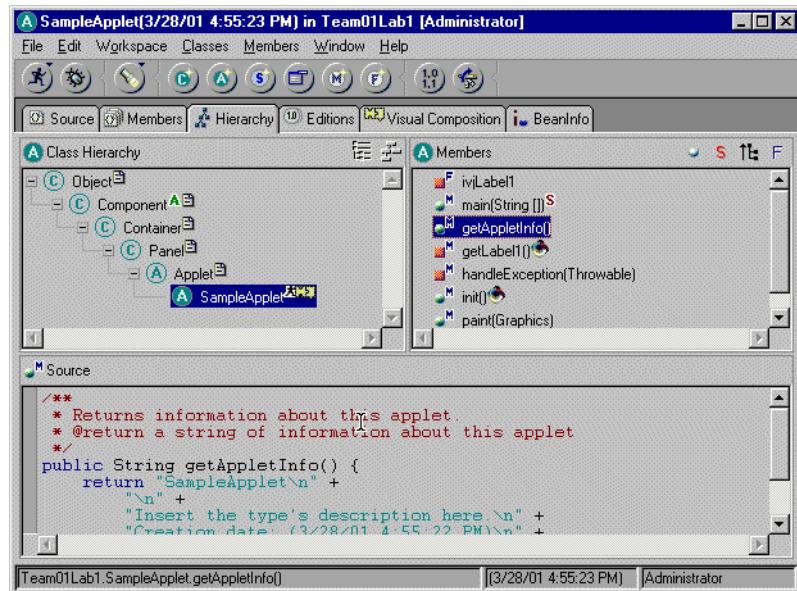


Figure 3-64 Sample applet

To run the applet, follow these steps:

1. From the applets pop-up menu, select **Run->Check Class Path....**
2. Switch to the Applet view of the property browser by clicking the **Applet** notebook tab.

Outside of the IDE, an HTML file is required to wrap the applet so it can run in a Web browser. The HTML file specifies the width, height, parameters, and so on of the applet. Within the VisualAge for Java IDE, this HTML file is not required since the settings are automatically made for you in the applets properties. You can still change the values proposed by the IDE on the Applet view of the property browser. Note that this window also provides an interface to add or change command line arguments, properties, and classpath information (Figure 3-65).

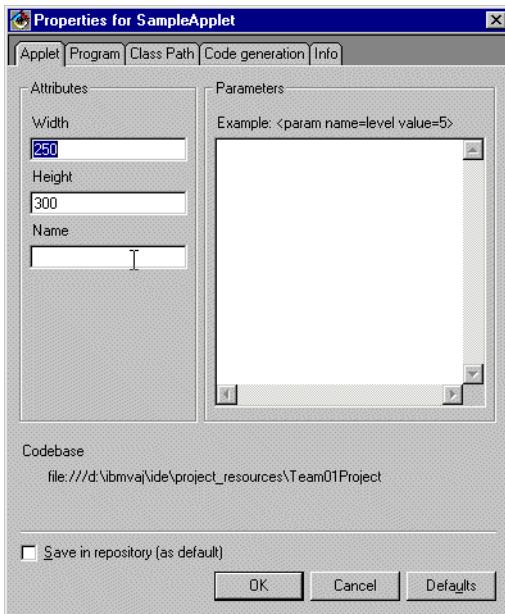


Figure 3-65 Applet properties example

3. Click **OK** to close the properties window.

### 3.2.5 Editor, debugger, and SmartGuides

In an object-oriented application development environment, developers need to perform many similar tasks as procedural developers. In addition, they perform a number of different tasks as part of a Rapid Application Development process. Specific to Java, these tasks include add a project, package, or class interactively.

A new project, package, or class can be added interactively. For example, a new class can be created from many different places in the IDE including the Workbench, Project Browser, Package Browser, and so on. Note the following tasks:

- ▶ **Add or change a method**

Adding or changing a method is probably the most important task of an application developer since this is the code that is actually executed in the running application. VisualAge for Java provides the ability to change a method at virtually any point. All browsers allow method source editing, and the debugger also allows methods to be added and edited.

- ▶ **Evaluate an expression**

Wherever a method can be entered or edited, an expression can be evaluated. For example, a developer may write a complex, concatenated line of Java code that needs to be tested. Instead of running the complete application, in many cases, VisualAge for Java allows the code snippet to be highlighted and run as is (provided it is a stand-alone piece of code).

For example, when debugging a method, the following code can be entered in the method pane, selected, and run:

```
System.out.println("Hello World!")
```

*Hello World* is displayed on the console window (the standard output device of the IDE).

► **Invoke methods**

As previously discussed, most code can be evaluated “on the spot” without running an application. From this, most methods can also be evaluated/invoked on the spot.

► **Test, debug, set breakpoints**

The debugger within the IDE is a powerful aid to the developer. It enables breakpoints to be set, hop over or into methods, run methods to completion, interactively patch code, and add new method classes while the running thread is held.

► **Patch code**

As previously stated, code can be patched at any time within the development cycle without losing the original code. This includes patching running code that may have caused the debugger to be invoked.

► **Compile class/method incrementally**

Outside of the IDE, a developer must modify the class as a complete unit. Therefore, if only one line of a method needs modifying, then the entire .java file needs to be edited and compiled. Within the VisualAge for Java IDE, individual methods can be edited and saved incrementally without the need to compile again the entire class that contains the method being changed.

► **Maintain project database**

The team development environment was introduced earlier in this chapter. This environment provides a complete project database for the development team.

► **Syntax check code**

VisualAge for Java detects syntax errors that occur when code violates Java syntax rules. For example, if you misspell a keyword or forget a semicolon, a message dialog box informs you of the type of syntax error when you try to save the code. In addition, the input cursor in the Source pane automatically selects the piece of code that caused the problem. Better than that, it even makes a proposal for the correction by giving you a list of known names to choose from if there are any available (Figure 3-66).

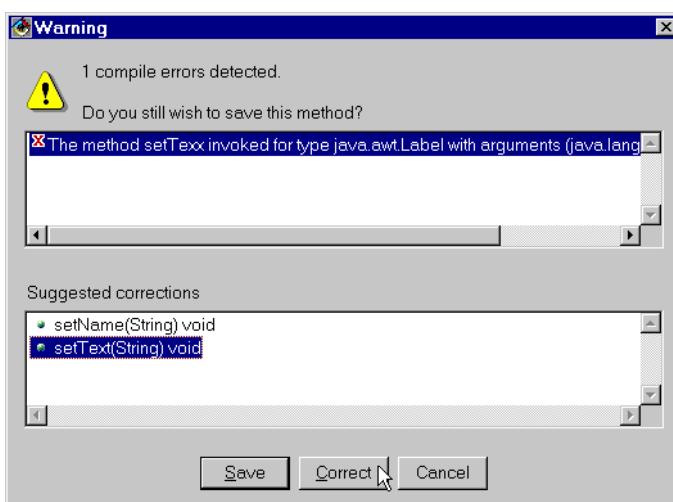


Figure 3-66 Syntax error suggested corrections example

## The editor pane

The editing pane (also called the Method Source pane) allows the developer to:

- ▶ Perform editing operations
  - ▶ Undo/Redo
- This option is accessed from the Edit menu item.
- ▶ Search in the workspace (image) for highlighted text

A developer can highlight some text and select Search from the pop-up menu to search the workspace, hierarchy, and working set for references to, or declarations of, the highlighted text or both. If you select the Working Set option, you can limit the search to a defined group or projects, or packages or classes. The search string can be declared as Type, Field, Constructor, or Method (Figure 3-67).

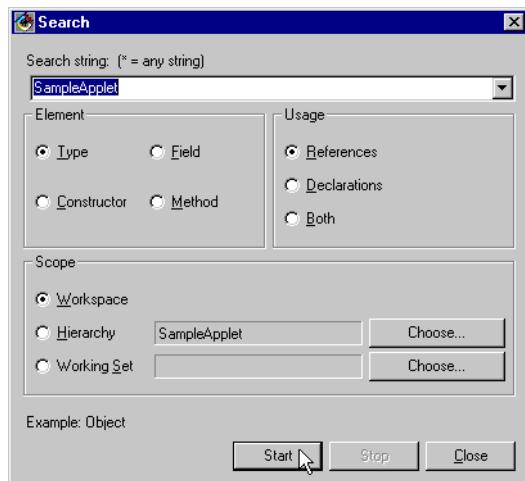


Figure 3-67 Search example

- ▶ Insert and remove breakpoints for debugging

A breakpoint is inserted or removed by moving the cursor to the left margin of the line requiring a breakpoint and double-clicking. In the IDE, this forces the debugger window to appear just before running this line.

- ▶ Save your changes

When changes are saved for a method, the entire method is syntax checked before it is saved. At any time, the previous version can be restored.

- ▶ Cancel your changes

If changes are made to a method and the developer selects another method to change without saving the pending changes, a warning dialog is displayed asking whether the pending changes should be saved.

## IDE setup

The IDE has some default settings and these can be modified by selecting **Window->Options...** from the workbench window (Figure 3-68).

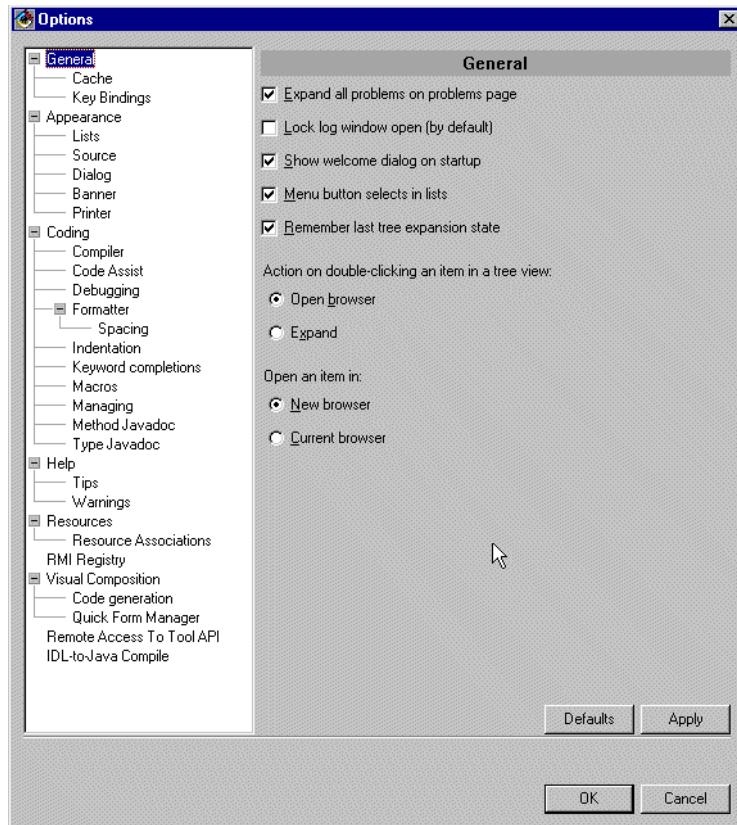


Figure 3-68 VisualAge for Java Options window

## The debugger

In addition to the features mentioned in “Debugging, setting breakpoints, and changes ‘on the fly’” on page 103, we discuss other interesting aspects of the debugger in VisualAge for Java. As you work in the IDE, you do not need to launch a special debugger virtual machine or start the virtual machine in the debug mode. The debugger opens automatically when you need it. It opens when:

- ▶ Execution hits a breakpoint that you inserted.
- ▶ An uncaught exception occurs.
- ▶ You select the debug smarticon on any tool bar.

You can use the debugger to step through code and inspect and change variables. You can also fix a bug by modifying the source from within the debugger.

VisualAge activates the debugger when one of a program's threads encounters a breakpoint. The top left pane (All Programs/Threads) displays the current thread that was created when you started the applet/application and the debugger invoked for whatever reason. In VisualAge, you create a thread (or multiple threads) whenever you run a program or evaluate code in the scrapbook. When the debugger opens on a breakpoint, the threads pane displays the thread that caused the debugger to open. The entry consists of an internal identifier for the thread and an indication of what caused the debugger to open. Displayed under the thread is its program stack from which each entry corresponds to a method that was called. Program stacks are in reverse chronological order (the most recent method is the top entry). The debugger lets you manipulate thread execution by dropping to a particular method. This is particularly useful if the debugger opens on an uncaught exception, since it lets you back up and repeat the steps that caused the exception to be thrown (see Figure 3-56 on page 105).

### **Stepping through the methods**

With the navigation buttons of the debugger, you can step through the current method. You can use the buttons to process the current statement (which is the one that is automatically selected), step into it, execute until the method returns, or resume processing the thread. When the debugger opens on a breakpoint, all the navigation buttons are enabled. By contrast, if the debugger opens because of an uncaught exception, the navigation buttons are disabled because the current process hit a dead end. In this case, you must first drop the program stack entry that throws the exception to reset the current status of processing. The options include:

- ▶ **Step Into**

Steps into the current statement and invokes the method (if any). A new program stack entry is added to the list, and the Source pane displays the source of the method that you stepped into. Use this smarticon to follow a method and determine what it does.

- ▶ **Step Over**

Executes the statement that is currently selected in the Source pane. The values of local variables are updated.

- ▶ **Run to Return**

Executes all statements in the method that is currently selected in the All Programs/Threads pane until the method is about to return and stops. All local variables are updated.

- ▶ **Resume**

Continues processing. Select this smarticon to continue running the program. If the program is resumed successfully, its thread is removed from the debugger.

- ▶ **Suspend**

To examine a thread at any point while it is running, you must suspend it manually by selecting this smarticon. Then, you can modify or step through its methods and inspect its variables.

- ▶ **Terminate**

When you terminate a thread, it is removed from the debugger browser and cannot be suspended or resumed any more. The thread is terminated. To restart the thread, you must restart the program from the beginning.

### **Inspectors**

You can use an inspector to view the state of objects or variables that hold objects. With the inspector, you can:

- ▶ Inspect the result of evaluating a code fragment in the scrapbook or in the Variables pane of the debugger.
- ▶ Open a browser on the declarations of an object's class.
- ▶ Evaluate code fragments in the context of an object.
- ▶ Change the value of an object.

Perform the following example (for the scrapbook window, refer to “Other VisualAge for Java windows” on page 118):

1. Open a Scrapbook page by selecting **Window->Scrapbook** from the Workbench window. Type the code in Example 3-2 into that page:

---

*Example 3-2 Code entered into the scrapbook*

---

```
String[][] info =  
    {{ "Red", "Number", "R of RGB" },  
     { "Green", "Number", "G of RGB" },  
     { "Blue", "Number", "B of RGB" }};  
return info;
```

---

2. Select all of the code (Ctrl+A), and click **Inspect smarticon** on the tool bar. See Figure 3-69.

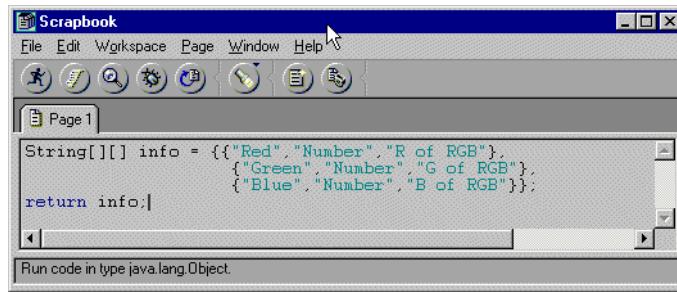


Figure 3-69 Scrapbook example

The inspector appears and shows the array object stored in the info variable. The title bar displays the identifier for the class of the inspected object, which is a two-dimensional String array. The title bar also shows the context from which you opened the inspector (from Page 1).

The Fields pane shows the elements of the array. The Value pane shows the value of a selected field.

The info array maps to a table with three rows and three columns (indexed 0 through 2). The top-level items in the Fields pane map to the three rows. By expanding items 0 through 2, you see that each row consists of three columns. Select the second row in the first column (info[1][0]).

It holds the parameter name Green. Internally, the string Green is represented as an array of characters that you can view in more detail by expanding the tree in the Fields pane. The icon to the left of the character array indicates that the internal representation is private (Figure 3-70).

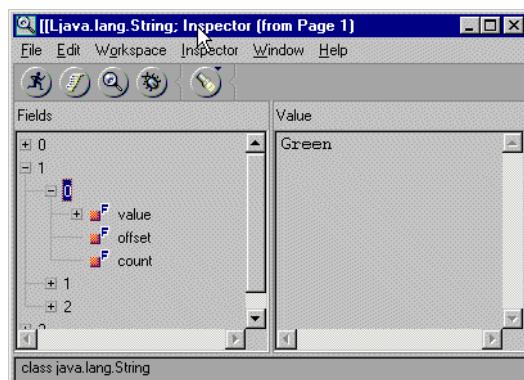


Figure 3-70 Inspector example

You can change the value of fields while you are inspecting an object by following these steps:

1. In the Fields pane, select the field that you want to modify.
2. In the Value pane, replace the text with the value that you want in the field.
3. Select **Save** from the pop-up menu.

The expression in the Value pane is evaluated. If the result can be assigned to the object, it is. When the code resumes, it uses the value. If the result cannot be assigned, the inspector displays an error message.

## **Other VisualAge for Java windows**

This section describes other windows available for VisualAge for Java.

### ***The Scrapbook window***

The Scrapbook helps you organize code fragments and notes. You can run any Java statement or expression from the scrapbook and control the context in which it is compiled.

To open the scrapbook, select **Scrapbook** from any window pull-down menu. The scrapbook appears with an empty page. From the scrapbook, you can run the code fragment or open an inspector on the object that is returned as the result of running the code. To open an inspector, select **Inspect** from the pop-up menu of the selected code fragment (see Figure 3-69 on page 117).

For example, most programming languages and environments take developers through the “Hello World” application as the first exercise in learning a new language or environment. With VisualAge for Java, this can be achieved in under a minute.

### ***Hello World in under a minute***

Complete the following steps:

1. Select **Window->Scrapbook**, and type this statement:

```
System.out.println("Hello World!");
```

2. Select the line of code that you typed.

3. Click the **Run** smarticon.

The console (the standard output device) appears and displays the string Hello World!. The code is automatically compiled by the built-in Java compiler and run by the built-in Java virtual machine.

### ***The console window***

The console is the standard output device (`System.out`) for Java programs that you run in VisualAge.

### ***The Repository Explorer***

With the Repository Explorer, you can explore the repository to view program components that are not present in the workspace/image.

Only within the Repository Explorer, you can define a solution that is a grouping of multiple projects. Solution map enables the loading of related project editions into the Workbench in one operation (Editions-> Add to workspace). This ensures that the correct version of related projects are loaded together.

### ***The log window***

The log displays messages and warnings from VisualAge.

## **SmartGuides and wizards**

The VisualAge for Java IDE comes with various SmartGuides (also known as wizards in other IDEs) that guide the developer through the repeatable process of creating a component.

For example, the Class Creation SmartGuide takes the developer through the standard process of creating a class including the following setup parameters:

- ▶ Which project is the class defined in?
- ▶ Which package is the class defined in?
- ▶ What is the class name?
- ▶ Which class is the super class?
- ▶ What happens when the SmartGuide completes?
  - Open a VCE (for example, if the class inherits from javax.swing.JFrame).
  - Open a class browser.
  - Do not open a browser.
  - Which interfaces (if any) does the class implement?
  - Which modifiers should be implemented?
    - Public
    - Abstract
    - Final
  - Should stub methods be generated?

There are a number of SmartGuides including class creation, interface creation, method creation, applet creation, and Fix/Migrate SmartGuide. The Fix/Migrate SmartGuide can repair broken class or package references due to renaming of package in the Java 2 SDK or the renaming of user-defined program elements, in the Professional Edition.

In VisualAge for Java Version 3.5 Enterprise Edition, you can find many different SmartGuides, such as the Create Program Call SmartGuide and Convert Display File SmartGuide in ET/400, SmartGuides for creating session beans for the EAB, and so on.

## **VisualAge for Java help**

The VisualAge for Java IDE comes with extensive documentation consisting of numerous HTML files and PDF files. These files are mostly stored locally but can also contain links to pages on the Internet. Therefore, it is advantageous to have access to the Internet while searching for information. Help for VisualAge for Java can be accessed by selecting a menu item from the Help menu. It starts your default browser and displays the page that you chose. Navigate through the various topics on the left window to gain an idea of what can be found where inside the documentation of VisualAge for Java. Try also the VisualAge for Java Search, which works efficiently for searching specific topic.

## **3.3 System requirements and prerequisites for version 3.5**

Table 3-2 on page 120 defines the system requirements for VisualAge for Java depending on which version you choose.

*Table 3-2 System requirements (Windows)*

Requirement or version	Professional	Enterprise
Hardware requirements	Intel Pentium II, or faster, compatible processor	Intel Pentium II, or faster, compatible processor
	48 MB of RAM (96 MB recommended) For Distributed Debugger: 128 MB RAM minimum 196 MB recommended	96MB of RAM (160 MB recommended) For Distributed Debugger: 128 MB RAM minimum 196 MB recommended
Hardware requirements	Hard disk space: 350 MB minimum, 400 MB or more recommended	Hard disk space: 400 MB minimum, 700 MB or more recommended
	CD-ROM drive	CD-ROM drive
	Mouse or pointing device	Mouse or pointing device
	Display: SVGA, 800x600 (1024x768 recommended)	Display: SVGA, 800x600 (1024x768 recommended)
Software requirements	Windows 98/2000 OR Windows NT 4.0 service pack 4,or later	Windows 98/2000 OR Windows NT 4.0 service pack 4,or later
Software prerequisites	A frames capable browser to access the HTML-based help and Web documentation such as: ► Netscape Navigator Version 4.7 or later ► Microsoft Internet Explorer Version 5.0 or later	A frames capable browser to access the HTML-based help and Web documentation such as: ► Netscape Navigator Version 4.7 or later ► Microsoft Internet Explorer Version 5.0 or later
	TCP/IP communications protocol configured and running	TCP/IP communications protocol configured and running
		ET/400 requires the OS/400 Release V4R3M0 or later
Supported languages (levels)	JDK Version 1.2.2	JDK Version 1.2.2

## 3.4 Summary

In summary, VisualAge for Java Version 3.5 is a member of the VisualAge family. It allows application developers to develop applications and Web-based applets, servlets, and JavaServer Pages using the Java language.

VisualAge for Java includes a powerful and full-function integrated development environment. The IDE is JDK 1.2.2 compliant. It allows the edit/compile/test of Java applications within the IDE prior to exporting the code for running in other JDK 1.2.2 compliant virtual machines and Web browsers. Because of its compliance with the JDK 1.2.2 API, the VisualAge for Java environment supports Java APIs for accessing remote components through the RMI and JDBC APIs. In particular, with the IBM Toolbox, the iSeries development environment is extremely rich. This enables access to the most common application development building blocks on the iSeries server (files, data queues, programs, and so on).

Because of the portability of JDK 1.2.2 compliant Java code, code that is developed using VisualAge for Java can run without change on the native iSeries Java virtual machine.

The IDE enables a developer to build and run applications and code snippets interactively without running the compile statement (`javac`) from the command line. All applications can run from within the IDE without needing to export the Java source or class files. This is achieved through the provision of a JDK 1.2.2 compliant Virtual Machine within the IDE. Because you can interactively modify code and run it without compilation, developers can debug code on the fly, spot errors in their code with the debugger, change it, and continue without bringing the running application down, all within the VisualAge for Java IDE.

VisualAge for Java is an open IDE. Developers can easily import and export Java source, class files, and JavaBeans, which may have been purchased by the company or made available on the Internet. The JavaBeans support in VisualAge for Java also enables a developer to import an existing JavaBean (for example, from the Internet) into VisualAge for Java, modify the bean, and export it again for use within another JDK 1.2.2 compliant development environment (for example, Symantic Cafe and Borland JBuilder).

VisualAge for Java has several components that extend its capabilities to make client/server programming easier. The Data Access Beans are beans provided with VisualAge for Java for developing programs that access relational database. The Enterprise Access Builder for Transactions provides components to access the function and data assets of your existing transaction systems. The IBM Toolbox for Java provides a series of classes specifically designed to access many iSeries features.





## IBM Toolbox for Java

The IBM Toolbox for Java provides Java classes that can be used to access iSeries resources. Since these classes are written in Java, they can be used on any Java-enabled platform. You can use them to build client/server applications that access iSeries resources from a client platform or to build Java applications that run on the iSeries server. This chapter covers the IBM Toolbox for Java. It discusses programming examples that highlight some of the key classes provided by the Toolbox.

## 4.1 Introduction to the IBM Toolbox for Java

The IBM Toolbox for Java is a series of Java classes that enables the Internet programming model. The classes can be used by Java applications, applets, servlets, and JavaServer Pages to access iSeries data and resources. The Toolbox does not require additional client support beyond what is provided by the Java virtual machine and JDK.

The IBM Toolbox for Java is currently available from IBM with OS/400 V4R2 or later as a no-charge licensed program product. It is a fully supported licensed program of the iSeries server. Table 4-1 shows the version of OS/400, the version of the Toolbox shipped, the minimum level of OS/400 that is supported as a server, and the minimum level of back server OS/400 to which the Toolbox for Java connects from the client.

Table 4-1 IBM Toolbox for Java versions

Shipped with OS/400	Toolbox shipped	JDK version supported	LPP details	Minimum OS level supported	Minimum OS back level supported
V4R2	Modification 0	1.1 or later	5763-JC1 V3R2M0	V3R2M0	V3R2M0
V4R3	Modification 1	1.1.1 or later	5763-JC1 V3R2M1	V3R2M0	V3R2M0
V4R4	Modification 2	1.1.6 or later	5769-JC1 V4R2	V4R2M0	V4R2M0
V4R5	Modification 3	1.2 or 1.1.x	5769-JC1 V4R5	V4R3M0	V4R2M0
V5R1	Modification 4	J2SE or 1.1.x	5722-JC1 V5R1	V4R4M0	V4R3M0

You can find additional support information at the IBM Toolbox for Java home page at:  
<http://www.series.ibm.com/toolbox>

The Toolbox provides support similar to functions available when using the Client Access APIs. It uses socket connections to the existing OS/400 servers as the access mechanism for the iSeries server. Each server runs in a separate job on the iSeries server and sends and receives architected data streams on a socket connection.

The IBM Toolbox for Java is delivered as a Java package that works with existing servers to provide an Internet-enabled interface to access and update iSeries data and resources.

The term *JTOpen* refers to the open source software product the IBM Toolbox for Java plus any additional enhancements provided by the open source community. JTOpen, which is governed by the IBM Public License, as well as its Java source code, is contained in the open source repository. For more information, see:  
<http://oss.software.ibm.com/developerworksopensource/jt400>

### 4.1.1 Installing the Toolbox

If you want to use the IBM Toolbox for Java classes inside the VisualAge for Java Integrated Development Environment, you must import these classes inside the IDE. VisualAge for Java Enterprise Edition and VisualAge for Java for iSeries simplifies this process. After you install VisualAge for Java 3.5 Enterprise Edition or VisualAge for Java for iSeries, the IBM Toolbox for Java Modification 3 classes are already available in the repository as part of the IBM Enterprise Toolkit for AS400 project. If you want to use the Toolbox classes, follow these steps:

1. From the workbench, click **File-> Quick Start**.
2. Click **Features->Add Feature**. Then, click **OK**.
3. Select **IBM Enterprise Toolkit**, and click **OK**.

This adds the Toolbox classes to your workspace. The IBM Enterprise Toolkit for AS/400 is listed under All projects.

The alternative is to use these steps:

1. Install the IBM Toolbox for Java LPP on an iSeries server, or access the Web site at:  
<http://www.iSeries.ibm.com/toolbox>
2. Download the classes (jt400.jar or jt400.zip) to your workstation.
3. Import the classes into the VisualAge for Java IDE.

### **Upgrading the IBM Toolbox for Java contained in VisualAge for Java 3.5**

The IBM Toolbox for Java version, currently shipped with VisualAge for Java Enterprise Edition Version 3.5, is modification 3. If you want to use the latest version (modification 4), you must upgrade VisualAge for Java. You should apply Patch 3 to VisualAge for Java prior to installing the new Toolbox support. The examples in this Redbook are all developed with the IBM Toolbox for Java Modification 4.

Once you successfully apply Patch 3, you can download the IBM Toolbox for Java Modification 4 from the iSeries server or the iSeries Web site. On the iSeries server, the Toolbox classes are found in the /qibm/ProdData/HTTP/Public/jt400/lib/ directory of the integrated file system.

The following instructions enable you to download and install the IBM Toolbox for Java Modification 4 into the VisualAge IDE:

1. Using Client Access or NetServer, map a network driver to the /QIBM directory of an iSeries V5R1 server.
2. Start IBM VisualAge for Java Enterprise Edition 3.5 or VisualAge for Java for iSeries.
3. Ensure that the IBM Enterprise Toolkit for AS/400 3.5 feature is loaded into the image.
4. Open the IBM Enterprise Toolkit for AS/400 V3.5 project.
5. Create an Open Edition of the project.
6. Delete the five packages:
  - com.ibm.as400.access
  - com.ibm.as400.data
  - com.ibm.as400.security
  - com.ibm.as400.security.auth
  - com.ibm.as400.vaccess

During the deletion, many error messages appear. Ignore them for now since they will be fixed when you load the new IBM Toolbox for Java.

7. Reselect the project containing the IBM Enterprise Toolkit for AS/400 package, select **File-> Import**.
8. Select the **Import from Jar** option, and click **Next**.
9. You are prompted for the file name of the JAR file to import. Locate the **JT400.Jar** file in the ProdData/HTTP/Public/jt400/lib/ directory of your mapped drive.
10. Select the **Overwrite existing resources without warning** option.
11. Select the **Version imported classes and new editions of packages/projects** option.  
The complete import SmartGuide should appear as shown in Figure 4-1 on page 126.

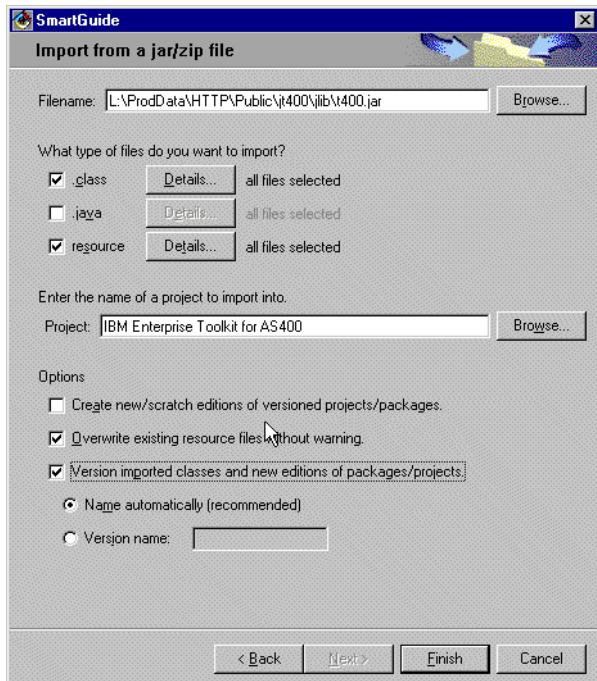


Figure 4-1 Importing the new Toolbox

12. Click the **Next** button to start the import process. This process is long running on all but the most powerful machine.
13. After the new IBM Toolbox for Java is versioned, you are prompted to add the new beans to a VCE folder. Select all the new beans, and add them to the AS/400 Toolbox folder as shown in Figure 4-2.

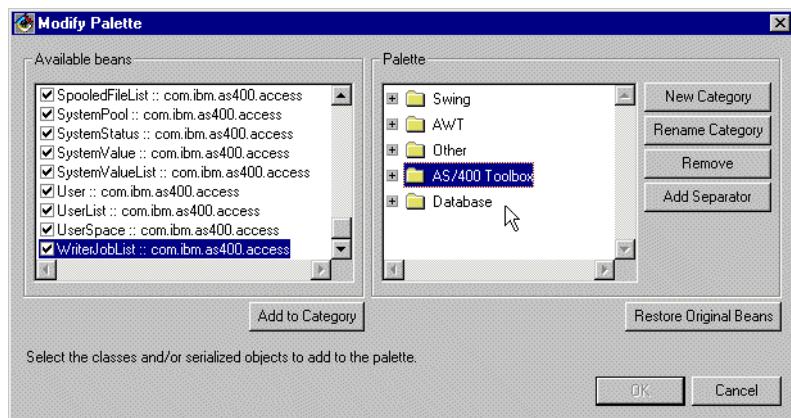


Figure 4-2 Adding the new iSeries beans to the VCE palette

14. Version the IBM Toolbox for Java 3.5 project.

You should now be able to use the IBM Toolbox for Java Modification 4 within IBM VisualAge for Java.

## 4.2 Enhancements

The IBM Toolbox for Java originally became available as part of OS/400 V4R2. A number of enhancements have been made to it. These enhancements are called *modifications*. This section covers the key features of the modifications.

### 4.2.1 V4R3 enhancements

With OS/400 V4R3, the IBM Toolbox for Java Modification 1 (5763-JC1 V3R2M1) offers two significant enhancements. Several new access classes are provided, as are the graphical user interface (GUI) classes. New classes are provided to access the following iSeries resources:

- ▶ **Digital certificates**

Digital certificates are digitally signed statements used for secured transactions over the Internet. Digital certificates can be used on iSeries servers running on Version 4 Release 3 and later. To make a secure connection using the secure sockets layer (SSL), a digital certificate is required.

- ▶ **Jobs**

The IBM Toolbox for Java jobs classes allow a Java program to retrieve the attributes of a job and list the active jobs. The job classes are:

- *Job*: Represents an iSeries job object
- *JobList*: Represents a list of iSeries jobs
- *JobLog*: Represents the job log of an iSeries server

- ▶ **Message queues**

The MessageQueue class allows a Java program to interact with an iSeries message queue. It acts as a container for the QueuedMessage class. The `getMessages()` method, in particular, returns a list of QueuedMessage objects. The MessageQueue class can perform these tasks:

- Set message queue attributes
- Get information about a message queue
- Receive messages from a message queue
- Send messages to a message queue
- Reply to messages

- ▶ **Queued messages**

The QueuedMessage class extends the AS400Message class. The QueuedMessage class accesses information about a message on an iSeries message queue. With this class, a Java program can retrieve:

- Information about where a message originated, such as the program, job name, job number, and user
- The message queue
- The message key
- The message reply status

- ▶ **Users and groups**

The user and group classes allow a Java program to get lists of users and groups on the iSeries server and information about each user. You use a UserList object to get a list of users and groups on the system. The only property of the UserList object that must be set is the AS400 object that represents the iSeries server from which the list of users is to be retrieved.

► **User space**

The UserSpace class represents a user space on the iSeries server. Required parameters are the name of the user space and the AS400 object that represents the iSeries server that has the user space. Methods exist in a user space class to perform the following functions:

- Create a user space
- Delete a user space
- Read from a user space
- Write to user space
- Get the attributes of a user space (a Java program can get the initial value, length value, and automatic extendible attributes of a user space.)
- Set the attributes of a user space (a Java program can set the initial value, length value, and automatic extendible attributes of a user space.)

With the GUI classes, you can visually represent your iSeries data and resources. Refer to 4.7, “Introduction to GUI component classes” on page 219, for more information and examples of using the GUI classes.

## 4.2.2 V4R4 enhancements

The IBM Toolbox for Java Modification 2 (5769-JC1 V4R2) adds significant enhancements and new features to the IBM Toolbox for Java. The following sections briefly summarize the enhancements.

### Additional or modified visual classes

The visual classes were enhanced to include:

► **Spooled file viewer**

The SpooledFileViewer is a new class that can be used to dynamically convert an iSeries spooled file into a graphical image that can be viewed on a client. However, be aware that this class requires the OS/400 level to be V4R4 or later and have the AFP Utilities licensed product installed.

► **Jobs**

VJobList and VJob objects can be added to iSeries panes (such as the AS400ExplorerPane) to display many different views of the jobs on a specified iSeries server.

► **Permission**

Permission classes enable a Java application to set or get various security information related to an iSeries object. The visual permissions information can be obtained with VIFSFile and VIFSDirectory classes through an iSeries plane.

► **System values**

Adding a VSystemValue object to an iSeries pane allows the user to view and change system values. Different views of system values can be obtained by adding them to different types of iSeries panes.

► **Users and groups**

Groups and individual users can be viewed and modified with the VUserAndGroup and VUserList classes. Some potentially useful methods are setUserInfo() and setGroupInfo(), which are available with VUsersAndGroup objects.

## **Additional or modified access classes**

The access classes were enhanced to include:

► **SSL support**

New for modification 2 is the ability for host servers to communicate using Secure Sockets Layer (SSL) support. To provide this function in Java, the IBM Toolbox for Java has a SecureAS400 object. SSL conversations can only take place between an IBM Toolbox for Java class and a V4R4 or later system that has SSL enabled Host Servers. For details on how to use this new support, see 13.3, "Securing applications with SSL" on page 506.

► **JDBC 2.0**

The JDBC 2.0 specification is a core part of Java 2 (JDK 1.2). By using JDBC 2.0 on a client, it is possible to use scrollable cursors to traverse a result set. In addition, JDBC 2.0 support enables an application to optimize data transfer from a database server. A JDBC 2.0 application can control the number of rows downloaded as a result of executing an SQL statement.

► **Data areas**

Using the DataArea and associated classes allow a Java application to set, retrieve, and monitor iSeries Data Area objects. It supports character, decimal, logical, and local data areas.

► **Message file**

Modification 2 allows programs to use iSeries message files. The MessageFile class allows you to receive a message from an iSeries message file. The MessageFile class returns an AS400Message object that contains the message.

► **Permission**

The permissions classes can be used to request and set permission information associated with iSeries IFS objects. For example, it is possible to list the user profiles that have explicit authority to a file or directory.

► **System status**

Using the SystemStatus and SystemPool classes, you can dynamically retrieve high-level information about the iSeries work management status. Using the SystemPool classes, you can even modify the pool sizes and other work management related values.

► **System values**

You can use SystemValue and SystemValueList objects to retrieve and set system value objects. iSeries security will always prevent an application from changing a value to which it is not authorized.

## **Additional new functions**

The IBM Toolbox for Java Modification 2 also offers these new functions:

► **Graphical Toolbox and Panel Definition Markup Language (PDML)**

The Graphical Toolbox is a tool and PDML is a powerful language that enables you to define panel layouts. PDML is an extension to XML. This way, you can avoid AWT and Swing programming.

► **Program Call Markup Language (PCML)**

PCML is an alternative way to call iSeries programs. It allows you to define an interface using PCML, which is another extension to XML. PCML simplifies calling iSeries objects since it performs operations such as the parameter conversions that are required when using the ProgramCall method. See *AS/400 XML in Action: PDML and PCML*, SG24-5959, for detailed information about PDML and PCML.

- ▶ **JarMaker and AS400ToolboxJarMaker**

These two classes are used to reduce the size of a JAR or ZIP file. The AS400ToolboxJarMaker is an extension to the JarMaker class. It knows about iSeries components, such as ProgramCall, so that specific components can be extracted from an IBM Toolbox for Java archive. Using these tools, you can reduce the size of deployment Java archives. See 13.2, “Java archive files” on page 500, for more information about JarMaker.

## Performance enhancements

Starting with modification 2, many of the iSeries classes have been coded to automatically detect if they are running directly on an iSeries server. With the exception of the JDBC and the IFS classes, the Toolbox classes can communicate more efficiently and make direct calls to existing iSeries APIs. Additional performance improvements can be made if the iSeries server Name property is set to localhost and the User ID and password is set to \*CURRENT. It is possible to use the setMustUseSockets(boolean) method to prevent or allow the direct calling of iSeries APIs.

### 4.2.3 V4R5 enhancements

The IBM Toolbox for Java Modification 3 originally shipped with OS/400, originally, also provides some new features and enhancements.

#### Additional access and visual classes

Modification 3 provides three new classes in the access and vaccess package:

- ▶ **FTP class**

The FTP class provides a programmable interface to FTP functions. You no longer have to use `java.runtime.exec()` or tell your users to run FTP commands in a separate application. That is, you can program FTP functions directly into your application.

- ▶ **JavaApplicationCall**

The JavaApplicationCall class provides you with the ability to easily run a Java program residing on the iSeries server from a client.

- ▶ **ServiceProgramCall**

The ServiceProgramCall class makes it possible for you to call an iSeries service program, pass data to an iSeries service program through input parameters, and access data the iSeries service program returns through output parameters.

#### Additional new functions

The IBM Toolbox for Java Modification 3 features the following new functions:

- ▶ **Security classes**

The IBM Toolbox for Java provides security classes. You use the security classes to provide secured connections to an iSeries server, verify a user's identity, and associate a user with the operating system thread when running on the local iSeries server.

- ▶ **HTML classes**

IBM Toolbox for Java HTML classes assist you in setting up forms and tables for HTML pages. The HTML classes implement the HTMLTagElement interface. Each class produces an HTML tag for a specific element type. The tag may be retrieved using the `getTag()` method and can then be embedded into any HTML document. The tags you generate with the HTML classes are consistent with the HTML 3.2 specification.

- ▶ **Servlet classes**

The servlet classes allow you to create servlets that can work with the access classes to access iSeries services.

- ▶ **Proxy support**

An alternative JAR file is now provided that works with a proxy server to provide similar functionality to the IBM Toolbox for Java classes, but with less download time. Proxy support allows the processing that the IBM Toolbox for Java needs to carry out for a task to be done on a Java virtual machine other than the Java virtual machine where the application actually is running.

- ▶ **System properties**

System properties can be specified to configure various aspects of the IBM Toolbox for Java. For example, you can use system properties to define a proxy server or a level of tracing. System properties are useful for convenient runtime configuration without needing to recompile code. System properties work like environment variables. If a system property is changed during runtime, the change will generally not be reflected until the next time the application is run.

### **Additional functions and features in Graphical Toolbox**

The Graphical Toolbox, introduced in V4R4, has been enhanced with more features and an improved look and feel. The changes made for V4R5 include:

- ▶ The PDML runtime environment now uses Swing 1.1. This allows you to take advantage of enhanced functions and performance available in the latest release of the Java Foundation Classes.
- ▶ The GUI Builder generates help in a composite file for easier development.
- ▶ The program call framework has been enhanced to provide PCML support for service program calls on the iSeries.

See *AS/400 XML in Action: PDML and PCML*, SG24-5959, for detailed information about PDML and PCML.

#### **4.2.4 V5R1 enhancements**

The IBM Toolbox for Java Modification 4 shipped with OS/400 V5R1 delivers many new features.

##### **New package**

IBM Toolbox for Java Modification 4 includes the new package com.ibm.as400.resource. This package provides a generic framework and consistent programming interface for working with various iSeries objects and lists.

##### **New classes**

The IBM Toolbox for Java Modification 4 also features many new classes in existing packages: access, html, servlet, and vaccess. The new classes enable you to:

- ▶ Convert text in bidirectional languages, such as Arabic and Hebrew, between iSeries and Java formats
- ▶ Manage a pool of AS400 objects to share and manage connections to your server, including JDBC connections
- ▶ Increase the variety of HTML tags you can include in your Java programs, including headings, meta tags, and lists

- ▶ Display IFS directory contents in an HTML list or a hierarchical tree
- ▶ Graphically present information about JDBC data sources and user-defined resources
- ▶ Display any resource in a servlet
- ▶ Use support for JDBC 2.0 Optional Package extensions
- ▶ Work with iSeries environment variables
- ▶ Request a license for a product installed on the iSeries
- ▶ Query and modify the state and configuration of the iSeries NetServer
- ▶ Ping the host servers from a command line or within a Java program

### **Enhanced classes**

The IBM Toolbox for Java Modification 4 also includes enhancements to existing classes. These enhancements offer:

- ▶ Improvements in IFS list processing
- ▶ Added methods to the CommandCall, ProgramCall, and ServiceProgramCall classes to:
  - Get the Job ID of the job actually running the command, program, or service program
  - Provide a way to directly call the command, program, or service program on the same thread as the application, instead of calling by way of a server (when running on the iSeries JVM)
  - Indicate if the iSeries command, program, or service program is thread-safe
- ▶ The ability to use SSL to encrypt connections between proxy clients and proxy servers
- ▶ User-defined grouping of system values
- ▶ An easier way to load and access additional message help text

### **New or enhanced functions**

The IBM Toolbox for Java Modification 4 also features new and enhanced functions:

- ▶ Faster string conversions are provided
- ▶ Better NLS support for system and error messages
- ▶ HTTP tunneling support for connecting between proxy clients and proxy servers

### **Additional functions and features in Graphical Toolbox**

The Graphical Toolbox incorporates these new features:

- ▶ The PDML framework now uses Java Help
- ▶ The GUI Builder now includes context-sensitive help
- ▶ Changes in the MessageViewer allow you to retrieve second level text for iSeries messages and include them in your own HTML detail help

## **4.3 IBM Toolbox for Java and host servers**

This set of interfaces provides the infrastructure needed to create and maintain socket connections to the iSeries servers, send and receive data streams, and handle a sign-on. This group of classes includes a private iSeries security manager class that maintains a list of validated iSeries servers and sign-on information for the system. These classes use the sign-on server and the central server to interact with the iSeries server. See Figure 4-3.

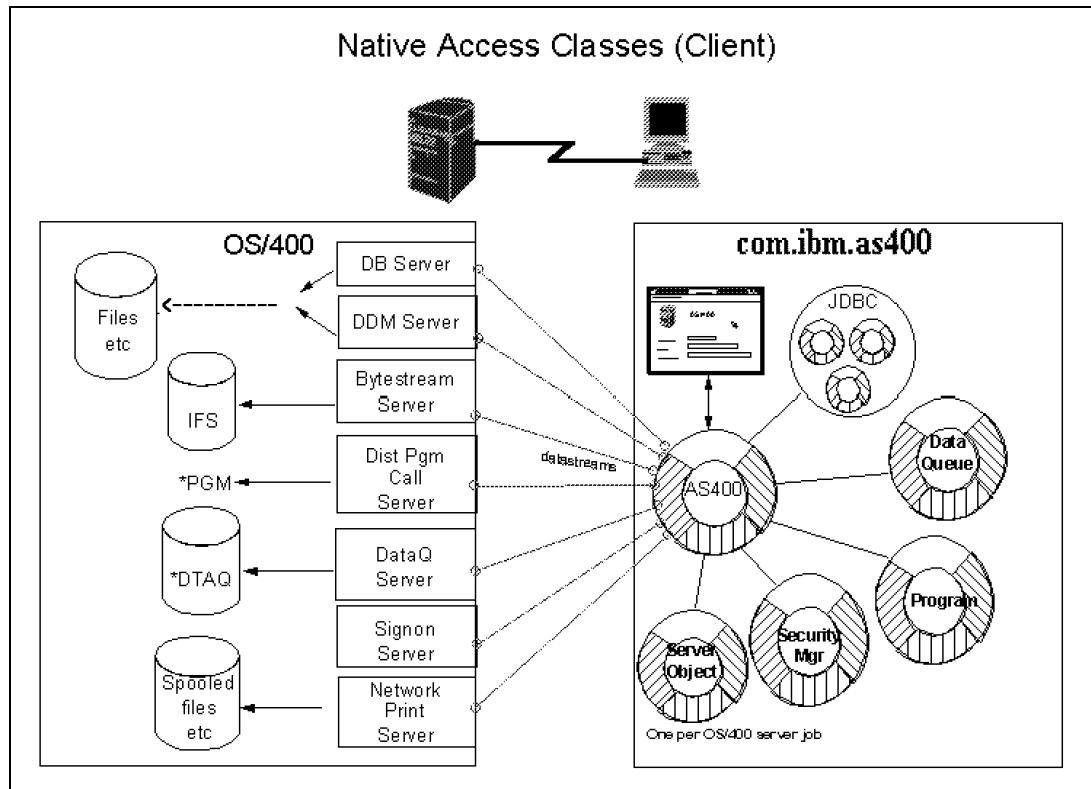


Figure 4-3 Java host server overview

The iSeries host servers must be running. To start the host servers, use the iSeries command:

```
STRHOSTSVR *ALL
```

When working with DDM, the TCP/IP server for DDM must be running. To start the TCP/IP server for DDM, use the iSeries command:

```
STRTCPSSVR *DDM
```

### 4.3.1 AS400 object, infrastructure, and sign-on

An AS400 object manages the following elements:

- ▶ A set of socket connections to the iSeries server
 

Each AS400 object contains one set of socket connections (up to one connection for each service type). This allows the Java programmer to control the number of connections to the iSeries server. To optimize communications performance, a Java program can create multiple AS400 objects for the same iSeries server. This allows multiple socket connections to the iSeries server. Java programs that want to conserve iSeries resources create only one AS400 object. This reduces the number of connections and reduces the amount of resources used on the iSeries server. You can also choose to use a connection pool to manage connections. This approach reduces the amount of time it takes to connect to the iSeries by reusing a connection previously established for the user.
- ▶ Sign-on behavior for the iSeries server
 

This includes prompting the user for sign-on information, password caching, and default user management.

- ▶ Prompting for sign-on information

Prompting for a user ID and password may occur when connecting to the iSeries server. Java programs can turn off prompting and graphical message windows displayed by the AS400 object. An example is an application running on a gateway on behalf of many clients. If prompts and messages are displayed on the gateway machine, the user has no way of interacting with the prompts.

- ▶ Password caching

To minimize the number of times a user has to type sign-on information, password caching can be used. The password cache applies to all AS400 objects that represent an iSeries server within a Java virtual machine. This means a cached password in one JVM is not visible to another virtual machine. The cache is discarded when the last AS400 object is destroyed. The sign-on dialog has a check box that gives the user the option to not cache any given password. When an AS400 object is constructed, the Java program has the option to supply the user ID and password. Passwords supplied on constructors are not cached.

- ▶ Default user management

To minimize the number of times a user has to sign on, a default user ID can be used. The default user ID is used when a user ID is not provided by the Java program. The default user ID can be set either by the Java program or through the user interface. If the default user ID is not established, the sign-on dialog allows the user to set the default user ID. Once the default user ID is established for a given iSeries server, the sign-on dialog does not allow the default user ID to be changed.

The Java program must provide an AS400 object when using an instance of a class that accesses the iSeries server. For example, the CommandCall object requires an AS400 object before it can send commands to the iSeries server.

## 4.4 The base API package

The base API package contains a set of Java classes that represent iSeries data and resources. The classes do not have an end-user interface. They simply move data back and forth between the client program and an iSeries server under the control of the client Java program. The Java classes in the base API package have these functional responsibilities:

- ▶ Describe the public interface for access to iSeries data and resources
- ▶ Manage a set of sockets connections to the server jobs
- ▶ Implement the public interface by creating and parsing the data streams defined for the appropriate server

Access to the following iSeries data and resources is provided:

- ▶ AS400 object, infrastructure, and sign-on
- ▶ JDBC access to DB2 UDB for iSeries data
- ▶ Record-level access to DB2 UDB for iSeries data
- ▶ Integrated file system
- ▶ Print functions
- ▶ Commands
- ▶ Program calls
- ▶ Data queues

The following functions do not directly access iSeries data and resources, but provide useful services for Java programmers accessing iSeries data:

- ▶ iSeries data types
- ▶ iSeries data description
- ▶ Access to iSeries messages generated from a command, program call, or print operation

#### 4.4.1 Data descriptions and conversions

The data conversion APIs provide the ability to convert numeric and character data between iSeries and Java formats. Conversion may be needed when accessing iSeries data from a Java program. The data conversion APIs support the conversion of various numeric formats and between various EBCDIC code pages and unicode.

Two levels of support are provided by the data conversion APIs:

- ▶ Data types convert data between the iSeries and Java format.
- ▶ Record-level conversion builds on data types to support converting all fields in a record with a single method call. The RecordFormat class allows the program to describe data that makes up a DataQueueEntry, ProgramCall parameter, record-level database access, or any buffer of iSeries data. The Record class allows the program to convert the contents of the record and access the data by field name.

#### 4.4.2 iSeries data types

Table 4-2 shows a set of classes, which represent iSeries data as Java data types to simplify the handling of iSeries data for Java programmers. Each class converts data between the iSeries representation and the Java representation of the data.

*Table 4-2 iSeries data types*

Class	Description
AS400Bin2	Provides a converter between a Short object and a signed two-byte binary number.
AS400Bin4	Provides a converter between an Integer object and a signed four-byte binary number.
AS400Bin8	Provides a converter between a Long object and a signed eight-byte binary number.
AS400UnsignedBin2	Provides a converter between an Integer object and an unsigned two-byte binary number.
AS400UnsignedBin4	Provides a converter between a Long object and an unsigned four-byte binary number.
AS400Float4	Provides a converter between a Float object and a four-byte floating point number.
AS400Float8	Provides a converter between a Double object and an eight-byte floating point number.
AS400PackedDecimal	Provides a converter between a BigDecimal or double object and a packed decimal format floating point number.
AS400ZonedDecimal	Provides a converter between a BigDecimal or double object and a zoned decimal format floating point number.
AS400Text	Provides character set conversion between Java String objects and iSeries native code pages.

Class	Description
AS400ByteArray	Provides a converter between a byte array and fixed-length byte array representing iSeries data that cannot be converted.
AS400Array	Provides a composite data type representing an array of AS400DataType objects.
AS400Structure	Provides a composite data type representing a structure of AS400DataType objects.
AS400JDBCBlob	Provides access to binary large objects.
AS400JDBCBlobLocator	Provides access to binary large objects.
AS400JDBCClob	Provides access to character large objects.
AS400JDBCClobLocator	Provides access to character large objects.
AS400JDBCInputStream	Provides access to binary data using an input stream.

#### 4.4.3 Record-level conversions

The classes shown in Table 4-3 allow Java programs to define iSeries data in a way that is similar to how data is defined on the iSeries server. These classes provide a way to define field descriptions and record formats, which are used to describe iSeries data. Data is accessed in a record object using field names defined by the field description object of the associated record format object.

Table 4-3 Record-level conversion

Public class	Description
Field Description	<p>Allow the Java program to describe the contents of a field or parameter with a data type and a string containing the name of the field. If the program is working with data from record-level access, it can also specify any iSeries data definition specification (DDS) keywords that further describe the field. The following field description classes are provided:</p> <ul style="list-style-type: none"> <li>▶ BinaryFieldDescription</li> <li>▶ CharacterFieldDescription</li> <li>▶ DateFieldDescription</li> <li>▶ DBCSEitherFieldDescription</li> <li>▶ DBCSGraphicFieldDescription</li> <li>▶ DBCSOnlyFieldDescription</li> <li>▶ DBCSOpenFieldDescription</li> <li>▶ FloatFieldDescription</li> <li>▶ HexFieldDescription</li> <li>▶ PackedDecimalFieldDescription</li> <li>▶ TimeFieldDescription</li> <li>▶ TimestampFieldDescription</li> <li>▶ ZonedDecimalFieldDescription</li> </ul>
RecordFormat	Allows the Java program to describe a group of fields or parameters. A Record object contains data described by a record format object. If the program is using record-level access classes, the record format class also allows the program to specify descriptions for key fields.
Record	Allows the Java program to process data described by the RecordFormat class. Data is converted between byte arrays containing the iSeries data and Java objects.

Public class	Description
LineDataRecordWriter	The class writes the record data, in line data format, to an OutputStream. The class translates the data into bytes by using the specified CCSID. The record format associated with the record determines the format of the data.

#### 4.4.4 JDBC specification

JDBC is an application programming interface, which is included in the Java platform, that enables Java programs to connect to a wide range of databases.

The IBM Toolbox for Java JDBC driver allows you to use JDBC API interfaces to issue Structured Query Language (SQL) statements to and process results from iSeries databases. The server also has a JDBC driver that is optimized for use on the server. When you run an application on the server, the Toolbox JDBC driver can automatically forward JDBC calls to the IBM Developer Kit for Java JDBC Driver. This usually results in better performance. You must set a driver property to enable this function.

#### Different versions of JDBC

Different versions of the JDBC API exist, and the IBM Toolbox for Java JDBC driver supports the following versions:

- ▶ JDBC 1.2 API (the `java.sql` package) is included in the Java Platform 1.1 core API and JDK 1.1.
- ▶ JDBC 2.1 core API (the `java.sql` package) is included in both the Java 2 Platform, Standard Edition (J2SE) and the Java 2 Platform Enterprise Edition (J2EE).
- ▶ JDBC 2.0 Optional Package API (the `javax.sql` package) is included in J2EE and is available as a separate download from Sun. These extensions were formerly named the JDBC 2.0 Standard Extension API.

#### Supported interfaces

Table 4-4 lists the supported JDBC interfaces and the API required to use them.

Table 4-4 Supported JDBC interfaces and the API required

Supported interface	Required API
<i>Blob</i> provides access to binary large objects (BLOBs)	JDBC 2.1 core
<i>CallableStatement</i> runs SQL stored procedures	JDK 1.1
<i>Clob</i> provides access to character large objects (CLOBs)	JDBC 2.1 core
<i>Connection</i> represents a connection to a specific database	JDK 1.1
<i>ConnectionPoolDataSource</i> represents a factory for PooledConnection objects	JDBC 2.0 Optional Package
<i>DatabaseMetaData</i> provides information about the database as a whole	JDK 1.1
<i>DataSource</i> represents a factory for database connections	JDBC 2.0 Optional Package
<i>Driver</i> creates the connection and returns information about the driver version	JDK 1.1
<i>PreparedStatement</i> runs compiled SQL statements	JDK 1.1

Supported interface	Required API
<i>ResultSet</i> provides access to a table of data that is generated by running a SQL query or DatabaseMetaData catalog method	JDK 1.1
<i>ResultSetMetaData</i> provides information about a specific ResultSet	JDK 1.1
<i>RowSet</i> is a connected row set that encapsulates a ResultSet	JDBC 2.0 Optional Package
<i>Statement</i> runs SQL statements and obtains the results	JDK 1.1
<i>XAConnection</i> is a database connection that participates in global XA transactions	JDBC 2.0 Optional Package
<i>XAResource</i> is resource manager for use in XA transactions	JDBC 2.0 Optional Package

## JDBC interface

The IBM Toolbox for Java implements the standard JDBC interface for access to data. JDBC defines a consistent set of classes and interfaces for communication with a database server.

The advantages of using JDBC are that it is an industry standard and it is easy to use. Being an industry standard allows the iSeries developer to use generic Java applets and applications and point them to the iSeries server for data storage and retrieval. Additionally, the Java developer can use the iSeries server as a server without worrying about iSeries specific implementation issues. JDBC can be easier to use than the other classes in the IBM Toolbox because the driver takes care of all data conversion issues. iSeries data types are automatically mapped to Java data types. The Java developer does not need to be concerned with the actual data representation on the iSeries server.

When using JDBC, you need to reference the following interfaces under the `java.sql` package and `javax.sql` package.

- ▶ **Driver:** Creates the connection and returns information about the driver version.
- ▶ **Connection:** Represents a connection to a specific database.
- ▶ **Statement:** Runs SQL statements and obtains the results.
- ▶ **PreparedStatement:** Runs pre-compiled SQL statements.
- ▶ **CallableStatement:** Runs SQL stored procedures.
- ▶ **ResultSet:** Provides access to a table of data generated by running an SQL statement or DatabaseMetaData catalog method.
- ▶ **ResultSetMetaData:** Determines the types and properties of the columns in a ResultSet.
- ▶ **DatabaseMetaData:** Provides catalog methods, which provide information about the database.
- ▶ **Blob:** The representation (mapping) in the Java programming language of an SQL BLOB. An SQL BLOB is a built-in type that stores a binary large object as a column value in a row of a database table.
- ▶ **Clob:** The representation (mapping) in the Java programming language of an SQL CLOB. An SQL CLOB is a built-in type that stores a character large object as a column value in a row of a database table.

Accessing data on the iSeries server using JDBC in your application involves the following steps:

1. Register the iSeries JDBC driver.
2. Connect to the database.
3. Define and prepare SQL statements.

4. Execute SQL statements.
5. Obtain and process results of the statements.
6. Close the statements.
7. Close the database connection.

## JDBC performance tips

JDBC from a Java program communicates with the same server program on the iSeries server as the Client Access ODBC driver. Any server side tuning suggestions for ODBC apply to JDBC. For more information on ODBC performance related issues, please refer to *AS/400 Client/Server Performance Using the Windows Client*, SG24-4526.

JDBC allows SQL statements to be sent to the iSeries server for execution. If an SQL statement is run more than one time, use a `PreparedStatement` object to execute the statement. A `PreparedStatement` compiles the SQL once, so that subsequent executions run quickly. If a plain `Statement` object is used, the SQL must be compiled and run every time it is executed. Use *Extended Dynamic support*. It caches the SQL statements in SQL packages on the iSeries server. Also turn on **package cache**, and cache SQL statements in memory.

Do not use a `PreparedStatement` object if an SQL statement is run only one time. Compiling and running a statement at the same time has less overhead than compiling the statement and running it in two separate operations.

Consider using JDBC stored procedures. In a client/server environment, stored procedures can help reduce communication I/Os, and therefore, help improve response time.

Use a *Just-In-Time (JIT) compiler* for your Java execution environment if possible. The latest JIT technology allows Java programs to perform almost as well as native code written in C or C++.

## JDBC properties

There are many properties that can be specified in the JDBC URL or in the JDBC properties object. Several of these properties can significantly affect the performance of a JDBC client/server application and should be used where possible. The properties control record blocking, package caching, and extended dynamic support. Selected properties and their settings are listed in Table 4-5. Other non-performance properties can be found in the Toolbox documentation.

**Note:** When using the properties for JDBC in your connection to the iSeries server, all property keywords and values have to be coded in lowercase letters (see the example in Table 4-5).

Table 4-5 General properties

Property	Description	Choices	Default
"user"	Specifies the user name for connecting to the iSeries server. If none is specified, the user is prompted, unless the "prompt" property is set to "false", in which case an attempt to connect will fail.	iSeries user	User is prompted
"password"	Specifies the password for connecting to the iSeries server. If none is specified, the user is prompted, unless the "prompt" property is set to "false", in which case an attempt to connect will fail.	iSeries password	User is prompted
"prompt"	Specifies whether the user should be prompted if a user name or password is needed to connect to the iSeries server. If a connection cannot be made without prompting the user, and this property is set to "false", an attempt to connect will fail.	"true" or "false"	"true"

Property	Description	Choices	Default
"secure"	Specifies whether SSL should be used.	"true" or "false"	"false"

See Table 4-6 through Table 4-10 on page 144 for a complete list of the JDBC properties (none of them are required).

*Table 4-6 Server properties*

Property	Description	Choices	Default
"libraries"	<p>Specifies the iSeries libraries to add to the server job's library list. The libraries are delimited by commas or spaces, and "*LIBL" may be used as a place holder for the server job's current library list. The library list is used for resolving unqualified stored procedure calls and finding schemas in databaseMetaData catalog methods. If "*LIBL" is not specified, the specified libraries replace the server job's current library list.</p> <p>In addition, if no default schema is specified in the URL, the first library listed in this property is also the default schema, which is used to resolve unqualified names in SQL statements.</p>	iSeries libraries	"*LIBL"
"transaction isolation"	Specifies the default transaction isolation.	"none", "read committed", "read uncommitted", "repeatable read", or "Serializable"	"none"

*Table 4-7 Format properties*

Property	Description	Choices	Default
"date format"	Specifies the date format used in date literals within SQL statements.	"mdy", "dmy", "ymd", "usa", "iso", "eur", or "jis"	(server job)
"date separator"	Specifies the date separator used in date literals within SQL statements. This property has no effect unless the "date format" property is set to "julian", "mdy", "dmy", or "ymd".	"/" (slash), "-" (dash), ":" (period), "," (comma), or "b" (space)	(server job)
"decimal separator"	Specifies the decimal separator used in numeric literals within SQL statements.	"." (period) or "," (comma)	(server job)
"naming"	Specifies the naming convention used when referring to tables.	"sql" (for example, schema.table), "system" (for example, schema/table)	"sql"
"time format"	Specifies the time format used in time literals within SQL statements.	"hms", "usa", "iso", "eur", or "jis"	(server job)

Property	Description	Choices	Default
"time separator"	Specifies the time separator used in time literals within SQL statements. This property has no effect unless the "date format" property is set to "hms".	":" (colon), "." (period), "," (comma), or " " (space)	(server job)
lob threshold	Specifies the maximum LOB (large object) size (in kilobytes) that can be retrieved as part of a result set. LOBs that are larger than this threshold will be retrieved in pieces using extra communication to the server. Larger LOB thresholds will reduce the frequency of communication to the server, but will download more LOB data, even if it is not used. Smaller LOB thresholds may increase the frequency of communication to the server, but will only download LOB data as it is needed.	"0" - "4194304"	"0"

Table 4-8 Performance properties

Property	Description	Choices	Default
"block criteria"	Specifies the criteria for retrieving data from the iSeries server in blocks of records. Specifying a non-zero value for this property reduces the frequency of communication to the server, and therefore, increases performance.  Ensure that record blocking is off if the cursor is going to be used for subsequent UPDATEs. Otherwise, the row that is updated may not necessarily be the current row.	"0" (no record blocking) "1" (block if FOR FETCH ONLY is specified) "2" (block unless FOR UPDATE is specified)	"2"
"block size"	Specifies the block size (in kilobytes) to retrieve from the iSeries server and cache on the client. This property has no effect unless the "block criteria" property is non-zero. Larger block sizes reduce the frequency of communication to the server, and therefore, increase performance.	"8", "16", "32", "64", "128", "256", or "512"	"32"
"prefetch"	Specifies whether to prefetch data upon executing a SELECT statement. This increases performance when accessing the initial rows in the ResultSet.	"true" or "false"	"true"
"extended dynamic"	Specifies whether to use extended dynamic support. Extended dynamic support provides a mechanism for caching dynamic SQL statements on the server. The first time a particular SQL statement is run, it is stored in an SQL package on the server. On subsequent runs of the same SQL statement, the server can skip a significant part of the processing by using information stored in the SQL package. If this is set to "true", a package name must be set using the "package" property.	"true" or "false"	"false"
"package"	Specifies the base name of the SQL package. Extended dynamic support works best when this is derived from the application name. Note that only the first seven characters are significant. This property has no effect unless the "extended dynamic" property is set to "true". In addition, this property must be set if the "extended dynamic" property is set to "true".	SQL package	""

Property	Description	Choices	Default
"package criteria"	Specifies the type of SQL statements to be stored in the SQL package. This can be useful to improve the performance of complex join conditions. This property has no effect unless the "extended dynamic" property is set to "true".	"default" (only store SQL statements with parameter markers in the package)  "select" (store all SQL SELECT statements to be stored in the package)	"default"
"package library"	Specifies the library for the SQL package. This property has no effect unless the "extended dynamic" property is set to "true".	Library for SQL package	"QGPL"
"package cache"	Specifies whether to cache SQL packages in memory. Caching SQL packages locally reduces the amount of communication to the server in some cases. This property has no effect unless the "extended dynamic" property is set to "true".	"true" or "false"	"false"
"package clear"	Specifies whether to clear SQL packages when they become full. Clearing an SQL package results in removing all SQL statements that have been stored in the SQL package. This property has no effect unless the "extended dynamic" property is set to "true".	"true" or "false"	"false"
"package add"	Specifies whether to add statements to an existing SQL package. This property has no effect unless the "extended dynamic" property is set to "true".	"true" or "false"	"true", "false"
"package error"	Specifies the action to take when SQL package errors occur. When an SQL package error occurs, the driver optionally throws an SQLException or post a warning to the Connection, based on the value of this property. This property has no effect unless the "extended dynamic" property is set to "true".	"exception", "warning", or "none"	"warning"

*Table 4-9 Sort properties*

Property	Description	Choices	Default
"sort"	Specifies how the server sorts records before sending them to the client.	<ul style="list-style-type: none"> <li>▶ "hex" (base the sort on hexadecimal values)</li> <li>▶ "job" (base the sort on the setting for the server job)</li> <li>▶ "language" (base the sort on the language set in the "sort language" property)</li> <li>▶ "table" (base the sort on the sort sequence table set in the "sort table" property)</li> </ul>	"job"
"sort language"	Specifies a three-character language ID to use for selection of a sort sequence. This property has no effect unless the "sort" property is set to "language".	Language ID	(locale)
"sort table"	Specifies the library and file name of a sort sequence table stored on the iSeries server. This property has no effect unless the "sort" property is set to "table".	Qualified sort table name	""
"sort weight"	Specifies how the server treats case while sorting records. This property has no effect unless the "sort" property is set to "language".	<ul style="list-style-type: none"> <li>▶ "shared" (upper- and lower-case characters are sorted as the same character)</li> <li>▶ "unique" (upper- and lower-case characters are sorted as different characters)</li> </ul>	"shared"

Table 4-10 Other properties

Property	Description	Choices	Default
"access"	Specifies the level of database access for the connection.	<ul style="list-style-type: none"> <li>▶ "all" (all SQL statements allowed)</li> <li>▶ "read call" (SELECT and CALL statements allowed)</li> <li>▶ "read only" (SELECT statements only)</li> </ul>	"all"
"errors"	Specifies the amount of detail to be returned in the message for errors that occur on the iSeries server.	"basic" or "full"	"basic"
"remarks"	Specifies the source of the text for REMARKS columns in ResultSets returned by databaseMetaData methods.	"sql" (SQL object comment) or "system" (OS/400 object description)	"system"
"translate binary"	Specifies whether binary data is translated. If this property is set to "true", the BINARY and VARBINARY fields are treated as CHAR and VARCHAR fields.	"true" or "false"	"false"
"trace"	Specifies whether trace messages should be logged. Trace messages are useful for debugging programs that call JDBC. However, there is a performance penalty associated with logging trace messages, so this property should only be set to "true" for debugging. Trace messages are logged to System.out.	"true" or "false"	"false"
"data truncation"	Specifies whether data truncation exceptions are thrown. If this property is set to "true", then data truncation exceptions are thrown if data needs to be truncated when writing to the database. If this property is set to "false", then no such data truncation exceptions are thrown. Either way, data truncation warnings are posted if data needs to be truncated when reading from the database.	"true" or "false"	"false"
"driver"	Specifies the JDBC driver implementation. The IBM Toolbox for Java JDBC driver chooses which JDBC driver implementation to use based on the environment. If the environment is an iSeries Java virtual machine on the same iSeries as the database to which the program is connecting, the native iSeries Developer Kit for Java JDBC driver is used. In any other environment, the IBMToolbox for Java JDBC driver is used. This property has no effect if the "secondary URL" property is set. This property cannot be set to "native" if the environment is not an iSeries Java virtual machine.	<ul style="list-style-type: none"> <li>▶ "default" (base the implementation on the environment)</li> <li>▶ "toolbox" (use the IBM Toolbox for Java JDBC driver)</li> <li>▶ "native" (use the iSeries Developer Kit for Java JDBC driver)</li> </ul>	"default"

#### 4.4.5 Record-level access

Record-level access supports Java applications, servlets, and Java applets when the programs and applets are running on an iSeries server that is at Version 4 Release 2 (V4R2) or later. If you are running to an iSeries server that is at an earlier level (V3R2, V3R7, or V4R1), record-level access supports only Java applications. Applets do not work.

iSeries physical files can be accessed one record at a time using the public interface of these classes. Files and members can be created, read, deleted, and updated. The record format can be defined by the programmer at application development time. Or, the format can be retrieved at runtime by the IBM Toolbox for Java support. These classes use the DDM server to access the iSeries server.

The classes in Table 4-11 are defined and implemented.

*Table 4-11 Record-level access classes*

Public class	Description
AS400File	Abstract base class for the record-level access classes. It provides the methods for sequential record access, creation and deletion of files and members, and commitment control activities.
SequentialFile	Represents an iSeries file whose access is by record number.
KeyedFile	Represents an iSeries file whose access is by key.
AS400FileRecordDescription	Provides the methods for retrieving the record format of an iSeries file.

#### 4.4.6 Integrated file system

The integrated file system classes allow a Java program to access files in the iSeries integrated file system as a stream of bytes or a stream of characters. The integrated file system classes were created because the java.io package does not provide file redirection and other iSeries functionality.

The function that is provided by the IFSFile classes is a superset of the function provided by the file IO classes in the java.io package. All methods in java.io FileInputStream, FileOutputStream, and RandomAccessFile are in the integrated file system classes. These classes use the bytestream server to access the iSeries server (see Table 4-12).

*Table 4-12 Integrated file system classes*

Public class	Description
IFSFile	Represents an object in the iSeries integrated file system. Similar to java.io.File
IFSJavaFile	Represents a file in the integrated file system (extends java.io.File)
IFSInputStream	Used to read (open an output stream on) a file in the integrated file system. Similar to java.io.FileInputStream
IFSTextInputStream	Represents a stream of character data being read from a file
IFSOutputStream	Used to write (open an output stream on) a file in the integrated file system. Similar to java.io.FileOutputStream
IFSTextOutputStream	Represents a stream of character data being written to a file

Public class	Description
IFSRandomAccessFile	Allows reading and writing of data from or to any specified location of a file in the integrated file system. Similar to java.io.RandomAccessFile
IFSFileDialog	Allows the user to move within the file system and to select a file within the file system

#### 4.4.7 Print

Print objects include spooled files, output queues, printers, printer files, writer jobs, and Advanced Function Printing (AFP) resources, which include fonts, form definitions, overlays, page definitions, and page segments. AFP resources are accessible only on Version 3 Release 7 (V3R7) and later iSeries servers.

**Note:** Trying to open an AFPResourceList to a system that is running an earlier version than V3R7 generates a RequestNotSupportedException exception.

The IBM Toolbox for Java classes for print objects are organized on a base class, PrintObject, and on a subclass for each of the six types of print objects. The base class contains the methods and attributes common to all iSeries print objects. The subclasses contain methods and attributes specific to each subtype.

*Table 4-13 Printer classes*

Public class	Description
PrintObject	An abstract base class for the various types of network print objects
AFPResource	Represents an iSeries AFP resource. An instance of this class can be used to manipulate an individual iSeries AFP resource
OutputQueue	Represents an iSeries output queue. An instance of this class can be used to manipulate an individual iSeries output queue (hold, release, clear, and so on)
Printer	Represents an iSeries printer. An instance of this class can be used to manipulate an individual iSeries printer
PrinterFile	Represents an iSeries printer file. An instance of this class can be used to manipulate an individual iSeries printer file
SpoooledFile	Represents an iSeries spooled file. You can use an instance of this class to manipulate an individual iSeries spooled file (hold, release, delete, send, read, and so on). To create new spooled files on the iSeries, use the SpoooledFileOutputStream class
WriterJob	Represents an iSeries writer job. An instance of this class can be used to manipulate an individual iSeries writer

#### 4.4.8 Command call

The CommandCall class allows a Java program to call a non-interactive iSeries command. Results of the command are available in a list of AS400Message objects. After the command runs, the Java program can use the getMessageList() method to retrieve any iSeries messages resulting from the command.

These classes use the Distributed Program Call server to access the iSeries server (Table 4-14)

*Table 4-14 CommandCall classes*

Public class	Description
Command Call	Represents an iSeries command object. This class allows the user to call an iSeries CL command.
AS400Message	Represents a message returned from an iSeries

#### 4.4.9 Program call

The ProgramCall class allows the Java program to call an iSeries program. You can use the ProgramParameter class to specify input, output, and input/output parameters. If the program runs, the output and input/output parameters contain the data that is returned by the iSeries program. If the iSeries program fails to run successfully, the Java program can retrieve any resulting iSeries messages as a list of AS400Message objects. These classes use the Distributed Program Call server to access the iSeries server (see Table 4-15).

*Table 4-15 Program call classes*

Public class	Description
ProgramCall	Allows a user to call an iSeries program, pass parameters to it (input and output), and access data returned in the output parameters after the program runs
ProgramParameter	Used with ProgramCall and ServiceProgramCall to pass parameter data to an iSeries program, from an iSeries program, or both
AS400Message	Represents a message returned from an iSeries

#### 4.4.10 Data queue

Both keyed and sequential data queues can be accessed using the public interfaces of the data queues classes. Entries can be placed on a data queue or removed. Data queues can be created or deleted on the iSeries server. These classes use the data queue server to access the iSeries server.

The public classes in Table 4-16 are defined and implemented.

*Table 4-16 Data queue classes*

Public class	Description
BaseDataQueue	Abstract class; represents an iSeries data queue object
DataQueue	Represents an iSeries data queue that is accessed sequentially
KeyedDataQueue	Represents an iSeries keyed data queue object
DataQueueEntry	Represents an entry on an iSeries data queue; holds the data for both keyed and sequential data queues

Public class	Description
KeyedDataQueueEntry	Represents an entry on an iSeries keyed data queue; holds additional data available when reading from a keyed data queue

#### 4.4.11 Proxy support

The IBM Toolbox for Java includes proxy support for some classes. Proxy support includes using the SSL protocol to encrypt data.

The proxy classes reside in the jt400Proxy.jar file, which ships with the rest of the IBM Toolbox for Java. The proxy classes, like the other classes in the IBM Toolbox for Java, comprise a set of platform independent Java classes that can run on any computer with a JVM. The proxy classes dispatch all method calls to a server application or proxy server. The full IBM Toolbox for Java classes are on the proxy server. When a client uses a proxy class, the request is transferred to the proxy server, which creates and administers the real IBM Toolbox for Java objects.

Some IBM Toolbox for Java classes are enabled to work with the proxy server application. These include:

- ▶ JDBC
- ▶ Record-level access
- ▶ Integrated file system
- ▶ Print
- ▶ Data queue
- ▶ Command call
- ▶ Program call
- ▶ Service program call
- ▶ User space
- ▶ Data area
- ▶ AS400 class
- ▶ SecureAS400 class

#### 4.4.12 Servlet support

The servlet classes that are provided with the IBM Toolbox for Java work with the access classes. They allow you to build servlets that can be deployed on a Java application server such as WebSphere Application Server.

*Table 4-17 Servlet classes that are available with the IBM Toolbox for Java*

Public class	Description
AuthenticationServlet	An HttpServlet implementation that performs basic authentication for servlets
AS400Servlet	An abstract subclass of AuthenticationServlet that represents an HTML servlet
ListRowData	Represents a list of data
RecordListRowData	Represents a list of records
SQLResultSetRowData	Represents an SQL result set as a list of data. This data is generated by an SQL statement through JDBC

Public class	Description
ListMetaData	Allows you get information about and change settings for the columns in a ListRowData class
RecordFormatMetaData	Allows you to provide the record format when you set the constructor's parameters or use the get and set methods to access the record format
SQLResultSetMetaData	Returns information about the columns of an SQLResultSetRowData object
StringConverter	An abstract class that represents a row data string converter
HTMLFormConverter	Extends StringConverter by providing an additional convert method called convertToForms()
HTMLTableConverter	Extends StringConverter by providing a convertToTables() method

## 4.5 How the iSeries server fits into this picture

The iSeries server can be a repository for data, programs, HTML documents, applets, servlet, and Java applications. An HTTP server running on the iSeries server can be used to serve Web pages and applets. The class files for Java applications can reside in the integrated file system of the iSeries server and accessed using a mapped drive.

When an HTML document containing an applet is served from the iSeries server, the class files are loaded from that iSeries server. The applet can access only that iSeries server. For applications, the class files are located using the CLASSPATH environment variable. On a network station (or comparable hardware), the CLASSPATH variable can be set to include the Toolbox class files. On a Windows (or other client operating system) workstation, there are two options. The workstation can have a mapped drive to the iSeries server (this requires Client Access), or the class files can be copied to the client. In either case, the CLASSPATH environment variable must be appropriately set to locate the class files.

No new function is needed on the iSeries server to use the IBM Toolbox for Java because the existing OS/400 servers are used. These servers are used:

- ▶ Database servers
- ▶ Distributed program call server
- ▶ Data queue server
- ▶ Network print server
- ▶ Bytestream server
- ▶ Sign-on server
- ▶ Central server
- ▶ DDM server

From the perspective of the IBM Toolbox for Java, the servers are a black-box interface to perform functions on the iSeries server. All requests directed toward data or resources on the iSeries server funnel through the servers.

## **4.5.1 Security**

Each connection to the iSeries server is validated for user ID and password. The Toolbox classes enable a single sign on for multiple connections to the iSeries server. Password expiration warnings and changing a password are supported.

iSeries security is enforced using the same security model as Client Access. The user must have a valid iSeries sign-on and proper authority to the AS400 objects or resources. A user ID and password are prompted if one is not provided by the user. The user ID and password are verified on the iSeries server. All passwords are encrypted prior to sending them to the iSeries server. To ensure additional security, passwords are not passed between classes except between the sign on GUI and the security class.

The servers run with the authorities of the user ID that is passed when the connection is made. No authorities are adopted.

## **4.5.2 National language support**

The IBM Toolbox for Java Modification 3 or later uses the internationalization support available with JDK 1.2. Translatable information resides in property files at runtime. Resource bundles are used to retrieve the proper text depending on the locale. Java internationalization support is built on Java locale objects that are defined by a country code, a language code, and a variant. The country codes are the two-letter ISO-3166 standard. The language codes are the two-letter ISO-639 standard. JDK 1.2 supports 105 different locales including both single-byte and double-byte locales, including both top-to-bottom and right-to-left languages.

The IBM Toolbox for Java does not attempt to add additional locales and is limited to the locales that Java supports.

Java supports a set of national languages, but it is a subset of the languages that the iSeries server supports. When a mismatch between languages occurs, for example, if you are running on a local workstation that is using a language that is not supported by Java, the IBM Toolbox for Java licensed program may issue some error messages in English.

Workstations that have the ability to display Arabic and Hebrew character sets also have the ability of right-to-left cursor movement. The IBM Toolbox for Java provides bidirectional application support. The AS400BidiTransform class provides layout transformations that enable you to convert bidirectional text in iSeries format (after first converting it to Unicode) to bidirectional text in Java format, or from the Java format to iSeries format.

You can specify a coded character set identifier (CCSID) for physical files. The CCSID describes the encoding scheme and the character set for character type fields contained in this file. The IBM Toolbox for Java incorporates its own conversion tables for over 100 commonly used CCSIDs. Previously, the IBM Toolbox for Java either deferred to Java for nearly all text conversion. If Java did not possess the correct conversion table, the IBM Toolbox for Java downloaded the conversion table from the server. The IBM Toolbox for Java performs all text conversion for any CCSID of which it is aware. When it encounters an unknown CCSID, it attempts to let Java handle the conversion. At no point does the IBM Toolbox for Java attempt to download a conversion table from the server.

This technique greatly reduces the amount of time it takes for an IBM Toolbox for Java application to perform text conversion. No action is required by the user to take advantage of this new text conversion; the performance gains all occur in the underlying converter tables.

### 4.5.3 Save and restore considerations

The Java classes and applets can reside in the IFS. There are no unique considerations for saving and restoring these entities.

### 4.5.4 Error recovery considerations

The Java model for error processing is to throw exceptions instead of returning return codes. The IBM Toolbox for Java follows this model. When an IBM Toolbox for Java class discovers an error, it throws an exception. Some exceptions contain a documented return code value. Some exceptions allow retrieving text that describes the error. The description for each IBM Toolbox for Java API includes a list of exceptions that can be thrown by the API. The application can catch these exceptions and handle them based on the API and exception returned.

The IBM Toolbox for Java handles errors so that the degree of success of an API is obvious to the application. This is a data integrity statement. The application knows the state of the data of an API call based on the exception (or lack of exception) generated.

In addition to throwing an exception, an error is logged to the IBM Toolbox for Java error log in some cases. An error is logged for unexpected conditions (for First Failure Support Technology (FFDC)), severe errors, and other places a message can help the user recover from the error. Errors are only logged if logging is turned on by the Java program.

### 4.5.5 Mapping iSeries data types to Java data types

Table 4-18 shows you how iSeries data types map to Java data types.

*Table 4-18 iSeries types mapped to Java types*

iSeries type	Java type
binary (1 – 4 digits)	short
binary (5 – 9 digits)	int
character	String
date	String
float (single precision)	float
float (double precision)	double
hex	byte
packed decimal	BigDecimal
zoned decimal	BigDecimal
time	String
timestamp	String

## 4.6 Using the access classes

The section covers application examples developed using IBM Toolbox for Java access classes. The IBM Toolbox for Java access classes provide functionality that is similar to using Client Access Express for iSeries APIs. However, Client Access Express for iSeries is not required for using the classes. The access classes use the existing iSeries servers as the access points to the iSeries server. Each server runs in a separate job on the iSeries and sends and receives data streams on a socket connection. See Figure 4-4.

We provide several examples including:

- ▶ iSeries database access:
  - JDBC
  - JDBC 2.0 Optional Package
  - JDBC stored procedures
  - DDM record-level access
  - Distributed Program Call
  - Data queues
- ▶ Network print
- ▶ Integrated file system
- ▶ Other classes, such as FTP classes

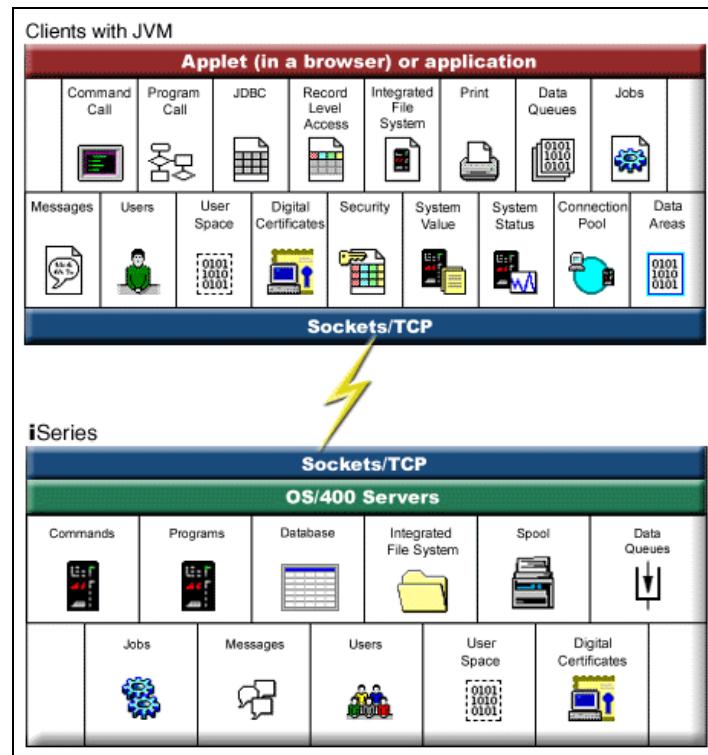


Figure 4-4 Access classes and iSeries servers

### 4.6.1 iSeries database access

The database access example applications shown in the remainder of this chapter are for the most part functionally equivalent. They allow for the retrieval, update, add, and delete of a record from a PARTS file on the iSeries server. All of these functions support the display of the entire PARTS file in a JTable. The PARTS file is defined as shown in Table 4-19.

Table 4-19 iSeries parts file

Field description	Length	Decimals	Type
Part number	5	0	Zoned
Description	25	0	Char
Quantity	5	0	Packed
Price	6	2	Packed
Part shipment date	10		Date

A start time and end time are updated on the window so that different iSeries access methods can be compared for performance. The examples were built using IBM VisualAge for Java development environment.

Both the Java code and the iSeries libraries are available for you to download from the Internet. See Appendix A, “Additional material” on page 529, for more information.

#### 4.6.2 JDBC application example

This example uses JDBC to access records in an iSeries database. The client program requests data from the iSeries database by sending SQL statements to the OS/400 host database server. The host server executes the SQL statement and returns the results to the client program in an SQL result set. The JDBC support handles all data conversions (see Figure 4-5).

A single part record can be retrieved, updated, added, and deleted, or all part records can be displayed in a JTable (Figure 4-6). Two classes drive the application: JDBCExample and JDBCExampleDisplayAll.

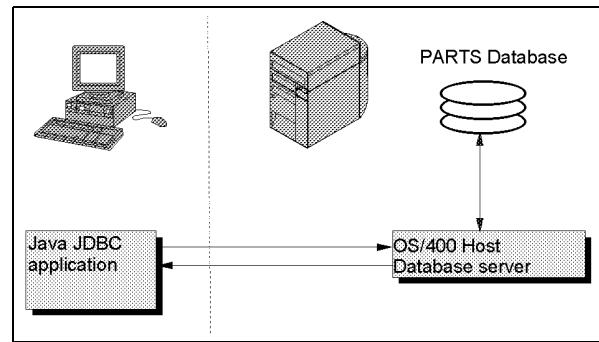


Figure 4-5 JDBC application

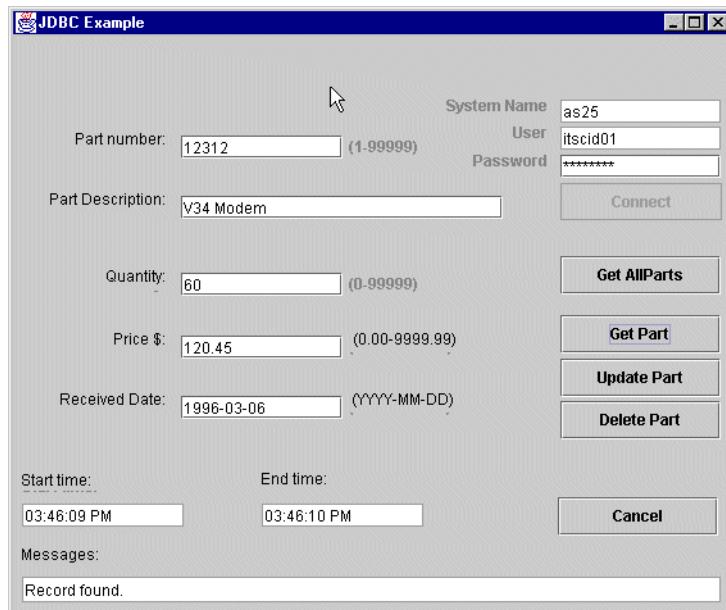


Figure 4-6 JDBCExample: A single part

The JDBCExample class creates the main application window, connects and disconnects the database, prepares the SQL statements, provides a GUI to display, and enters and manipulates the values for a single part record. All of the four basic database operations (Create, Read, Update, Delete) are implemented in this class. It instantiates a JDBCExampleDisplayAll object when the Get All Parts button is clicked (see Figure 4-7 on page 154).

Part#	Description	Qty	Price \$	Received
12301	50X Speed...	15	150.00	2001-09-09
12302	Ethernet P...	30	85.30	1995-12-17
12304	Ethernet P...	30	85.30	1995-12-17
12305	Home mou...	47	25.50	1996-02-18
12306	Gender-be...	75	8.50	1995-08-27
12307	600 dpi flat...	12	875.33	1996-03-01
12308	100 MHZ P...	4	1875.20	1996-02-24
12309	LaserJet T...	12	89.45	1995-12-17
12310	Logo mou...	376	7.25	1994-11-24
12312	V34 Mode...	60	120.45	1996-03-06
12313	Games joy...	32	42.75	1995-11-12
12314	3m printer ...	20	12.40	1996-01-23
12315	Anti-glare s...	45	34.77	1996-02-27
12316	Quad spee...	14	151.38	1996-01-12
12317	SCSI II Ca...	25	37.84	1995-11-13
12318	17 inch SV...	6	1388.59	1996-03-04
12319	Ethernet P...	30	107.60	1995-12-17
12320	Home mou...	47	32.16	1996-02-18

Figure 4-7 JDBCExample all parts

## JDBCExample class

This section investigates the key methods of the JDBCExample class.

### Instance variables

The following instance variables for accessing the database are declared for the class:

```
private java.sql.Connection dbConnect;
private java.sql.PreparedStatement psAllRecord;
private java.sql.PreparedStatement psSingleRecord;
private java.sql.PreparedStatement psUpdateRecord;
private java.sql.PreparedStatement psAddRecord;
private java.sql.PreparedStatement psDeleteRecord;
```

### The connectToDB method

The connectToDB method is called when the Connect button is clicked. String parameters representing the iSeries server name, user ID, and password are passed to the method. See Example 4-1.

---

#### Example 4-1 The connectToDB method example

```
public String connectToDB(String systemName, String userid, String password)
{
    try
    {
        setCursor(java.awt.Cursor.getPredefinedCursor(java.awt.Cursor.WAIT_CURSOR));
        java.sql.DriverManager.registerDriver(new com.ibm.as400.access.AS400JDBCDriver());
        dbConnect = java.sql.DriverManager.getConnection("jdbc:as400://" + systemName +
            "/apilib;naming=sql;errors=full;date format=iso;extended dynamic=true;" +
            "package=JDBCExa;package library=apilib", userid, password);
        psSingleRecord = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE PARTNO = ?");
        psAllRecord = dbConnect.prepareStatement("SELECT * FROM PARTS ORDER BY PARTNO");
        psUpdateRecord = dbConnect.prepareStatement("UPDATE PARTS SET PARTDS = ?, " +
            " PARTQY = ?, PARTPR = ?, PARTDT = ? WHERE PARTNO = ?");
        psAddRecord = dbConnect.prepareStatement("INSERT INTO PARTS (PARTDS, PARTQY, " +
            " PARTPR, PARTDT, PARTNO) VALUES(?, ?, ?, ?, ?)");
        psDeleteRecord = dbConnect.prepareStatement("DELETE FROM PARTS WHERE PARTNO = ?");
    }
```

```

        }
    catch (Exception e)
    {
        showException(e);
        setCursor(java.awt.Cursor.getPredefinedCursor(java.awt.Cursor.DEFAULT_CURSOR));
        return "Connect Failed.";
    }
    setCursor(java.awt.Cursor.getPredefinedCursor(java.awt.Cursor.DEFAULT_CURSOR));
    return "Connected to AS/400.";
}

```

---

**Note:** In the class descriptions throughout this chapter, the lines of code for each method precede the explanation. That is, the explanation itself follows with the bulleted text.

**Class:** JDBCExample

**Method:** connectToDB

Let's examine this method:

```
java.sql.DriverManager.registerDriver
(new com.ibm.as400.access.AS400JDBCDataSource());
```

- ▶ This statement loads the JDBC driver into the Java virtual machine. The fully-qualified name of the iSeries JDBC driver class is passed as a parameter:

```
dbConnect = java.sql.DriverManager.getConnection
("jdbc:as400://" + systemName + "/apilib;naming=sql;errors=full;date format=iso;extended
dynamic=true;package=JDBCEX;package library=apilib", userid, password);
```

- ▶ This statement creates a java.sql.Connection object called dbConnect. The form of the DriverManager's getConnection method used here has a URL, user ID, and password parameters. The URL is formatted.

```
jdbc:as400://systemName/defaultLibraryName;parameter1=value1;
parameter2=value2;...
```

- ▶ The default library name is optional, as are the properties. We use APILIB as the default library and specify the use of the ISO format for date fields. Error messages must contain all available information. We further specify some parameters for the performance properties to use extended dynamic support.

```
psSingleRecord = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE PARTNO =
?");
psAllRecord = dbConnect.prepareStatement("SELECT * FROM PARTS ORDER BY
PARTNO");
psUpdateRecord = dbConnect.prepareStatement("UPDATE PARTS SET PARTDS = ?, " +
" PARTQY = ?, PARTPR = ?, PARTDT = ? WHERE PARTNO = ?");
psAddRecord = dbConnect.prepareStatement("INSERT INTO PARTS (PARTDS, PARTQY, " +
" PARTPR, PARTDT, PARTNO) VALUES(?, ?, ?, ?, ?)");
psDeleteRecord = dbConnect.prepareStatement("DELETE FROM PARTS WHERE PARTNO =
?");
```

- ▶ These statements create five PreparedStatement objects. PreparedStatements are precompiled SQL statements that are more efficient to execute than plain Statements when run repeatedly. The "?" is used as a parameter marker, with the value set prior to running the PreparedStatement. The first statement creates an object that selects a

record from the parts file that has a PARTNO field equal to a value set at runtime. The second statement creates an object that selects all records from the parts file and orders the result set by the PARTNO. The third statement creates an object that selects a record from the parts file that has a PARTNO field equal to a value set at runtime. If that record is found, it is updated with the four other parameters also set at runtime. The fourth statement creates a new record in the parts file using five parameters to define the values of the fields that the new record contains. The fifth statement creates an object that selects a record from the parts file that has a PARTNO field equal to a value set at runtime. If that record is found, it is deleted from the parts file.

### **The *getRecord* method**

The *getRecord* method is called when the Get Part button is clicked. A string parameter containing the part number is passed, along with the four text field objects that are used to display values of other fields in the part record.

---

#### *Example 4-2 The *getRecord* method example*

---

```
public String getRecord(String partNo, javax.swing.JTextField partDesc,
javax.swing.JTextField partQty, javax.swing.JTextField partPrice, javax.swing.JTextField
partDate)
{
    java.sql.ResultSet rs = null;
    try
    {
        psSingleRecord.setInt(1, Integer.parseInt(partNo.trim()));
        rs = psSingleRecord.executeQuery();
        if (rs.next())
        {
            partDesc.setText(rs.getString("PARTDS").trim());
            partQty.setText(Integer.toString(rs.getInt("PARTQY")));
            partPrice.setText(rs.getBigDecimal("PARTPR").toString());
            partDate.setText(rs.getDate("PARTDT").toString());
        }
        else
        {
            partDesc.setText("");
            partQty.setText("0");
            partPrice.setText("0.00");
            partDate.setText("");
            return "Record not found.";
        }
    }
    catch (Exception e)
    {
        e.printStackTrace();
        showException(e);
        return "Error during SQL-Execution at SELECT.";
    }
    return "Record found.";
}
```

---

**Class:** JDBCExample

**Method:** getRecord

Let's explore this method:

```
java.sql.ResultSet rs = null;
```

- ▶ This line declares and initializes a variable, *rs*, to reference a ResultSet object.

```
psSingleRecord.setInt(1, Integer.parseInt(partNo.trim()));
```

- ▶ This line uses the preparedStatement method, *setInt*, to set the value of parameter 1 to the integer value of the part number passed on the parameter list. Note the *trim()* method, which cuts off leading and trailing blanks from the parameter *partNo*. Because of this, the statement does not throw an exception in case the user entered a part number with leading or trailing blanks.

```
rs = psSingleRecord.executeQuery();
```

- ▶ This line executes the SQL statement defined by the *psSingleRecord* PreparedStatement object and places the table of resulting records in a ResultSet object referenced by *rs*. The *executeQuery()* method always returns a result set.

```
if (rs.next()) {
```

- ▶ The *next()* method of the ResultSet object attempts to position the cursor of the result set to the next record from the result table. Because this is the first one read from the result set, the method positions to the first record from the result set and returns *true*. If there are no records to retrieve, the method returns a *false* value.

```
partDesc.setText(rs.getString("PARTDS").trim());  
partQty.setText(Integer.toString(rs.getInt("PARTQY")));  
partPrice.setText(rs.getBigDecimal("PARTPR").toString());  
partDate.setText(rs.getDate("PARTDT").toString());
```

- ▶ These lines retrieve values of database fields and place them in their corresponding screen fields. The ResultSet object has *getter* methods for many Java data types. Here we use the following methods:
  - **getString**: Returns the value of the column PARTDS as a String object.
  - **getInt**: Returns the value of the column PARTQY as an integer.
  - **getBigDecimal**: Returns the value of the PARTPR field as a BigDecimal object.
  - **getDate**: Returns the value of column PARTDT as a Date.

**Note:** In DB2 databases, it is likely that trailing blank characters are stored in the fields. In a GUI, we do not want these characters to be displayed, so the *trim()* method cuts them off from the database field PARTDS.

### ***The updateRecord method***

This method is called first when the Update/Add part button is clicked. String parameters containing the values of all entry fields on the screen are passed in. These values are used to update the part record designated by the value of the parameter part number. If the record does not exist yet, the *addRecord* method is run. This method does not have to return a result set. We are only interested in the fact of whether the database operation was successful. Instead of the *executeQuery()* method, the *executeUpdate()* method is used, which returns the number of database records affected by the SQL statement.

---

*Example 4-3 The updateRecord method example*

---

```
updateRecord(String partNo, javax.swing.JTextField partDesc, javax.swing.JTextField
partQty, java.swing.JTextField partPrice, javax.swing.JTextField partDate)
{
    try
    {
        psUpdateRecord.setString(1, partDesc.getText().trim());
        psUpdateRecord.setInt(2, Integer.parseInt(partQty.getText().trim()));
        psUpdateRecord.setFloat(3, new
            java.lang.Float(partPrice.getText().trim()).floatValue());
        psUpdateRecord.setString(4, partDate.getText().trim());
        psUpdateRecord.setInt(5, Integer.parseInt(partNo.trim()));
        int rowsUpdated = psUpdateRecord.executeUpdate();
        if (rowsUpdated > 0)
        {
            return java.lang.String.valueOf(rowsUpdated) + " Record updated.";
        }
        else
        {
            return addRecord(partNo, partDesc, partQty, partPrice, partDate);
        }
    }
    catch (Exception e)
    {
        e.printStackTrace();
        showException(e);
        return "Error during SQL-Execution at UPDATE.";
    }
}
```

---

**Class:** JDBCExample

**Method:** updateRecord

Let's break apart this method:

```
psUpdateRecord.setString(1, partDesc.getText().trim());
```

- ▶ This line uses the preparedStatement method, setString, to set the value of parameter 1 to the string value of the part description passed on the parameter list. As before, the trim() method cuts off leading and trailing blanks.

```
int rowsUpdated = psUpdateRecord.executeUpdate();
```

- ▶ This line runs the SQL update statement and returns the number of rows affected, which can be used for validation whether the update was successful. Unlike the executeQuery() method, the executeUpdate method does not return a result set.

```
return addRecord(partNo, partDesc, partQty, partPrice, partDate);
```

- ▶ In case that the update was not successful (rows Updated = zero), the addRecord method is called and returns its result instead of the update method.

Instead of using a prepared statement, you can create a dynamic SQL statement object to perform the same task. Performance improves if the PreparedStatement is used, assuming the update occurs more than one time during the user session. The ad hoc SQL statement appears as shown in Example 4-4.

---

*Example 4-4 The updateRecord method dynamic SQL example*

---

```
java.sql.Statement sUpdateRecord = dbConnect.createStatement();
String updatestring = "UPDATE PARTS SET PARTDS = '" + partDesc.getText() + "', PARTQY = " +
partQty.getText() + ", PARTPR = " + partPrice.getText() + ", PARTDT = '" +
partDate.getText() + "' WHERE PARTNO = " + partNo;
int rowsUpdated = 0;
try
{
    rowsUpdated = sUpdateRecord.executeUpdate(updatestring);
}
catch (java.sql.SQLException SSQL)
if (rowsUpdated > 0)
{
    return java.lang.String.valueOf(rowsUpdated) + " Record updated.";
}
```

---

```
java.sql.Statement sUpdateRecord = dbConnect.createStatement();
```

- ▶ This line creates a dynamic SQL statement object called sUpdateRecord.

```
String updatestring = "UPDATE PARTS SET PARTDS = '" + partDesc.getText() + "', PARTQY = " +
partQty.getText() + ", PARTPR = " + partPrice.getText() + ", PARTDT = '" +
partDate.getText() + "' WHERE PARTNO = " + partNo;
```

- ▶ These lines build a String value for the update SQL statement. Standard SQL syntax is used to update part fields with values passed on the parameter list for the part number requested.

```
rowsUpdated = sUpdateRecord.executeUpdate(updatestring);
```

- ▶ This line runs the SQL update statement and returns the number of rows affected, which can be used for validation whether the update was successful.

### **The addRecord method**

This method works basically the same way as the updateRecord method, except that it uses a different preparedStatement (INSERT).

### **The deleteRecord method**

This method works basically the same way as the updateRecord method, except that it uses a different preparedStatement (DELETE).

### **The dispose method**

This method is called when the application window is closed.

---

*Example 4-5 The dispose method example*

---

```
public void dispose()
{
    try
    {
        if (psSingleRecord != null)
        {
            psSingleRecord.close();
        }
        if (psAllRecord != null)
        {
            psAllRecord.close();
        }
    }
}
```

```

        }
        if (psUpdateRecord != null)
        {
            psUpdateRecord.close();
        }
        if (psAddRecord != null)
        {
            psAddRecord.close();
        }
        if (psDeleteRecord != null)
        {
            psDeleteRecord.close();
        }
        if (dbConnect != null)
        {
            dbConnect.close();
        }
    }
    catch (Exception e)
    {
        System.out.println("Dispose Exception" + e);
    }
    this.getComponents();
    if (ivjJDBCExampleDisplayAll1 != null)
        ivjJDBCExampleDisplayAll1.dispose();
super.dispose();
System.exit(0);
return;
}

```

---

**Class:** JDBCExample

**Method:** dispose

Let's examine this method:

```

if (psSingleRecord != null)
{
    psSingleRecord.close();
}

```

- ▶ These lines release the psSingleRecord PreparedStatement database and JDBC resources immediately. The current ResultSet is closed as well. The PreparedStatements should be tested for null, since we do not know whether the user is connected to the database (closing a PreparedStatement, which is null, throws an exception).

```
dbConnect.close();
```

- ▶ This line disconnects from the iSeries server.

```
if (ivjJDBCExampleDisplayAll1 != null) ivjJDBCExampleDisplayAll1.dispose();
```

- ▶ If the JDBCExampleDisplayAll class is still instantiated, this line calls its dispose method to shut it down.

```
super.dispose();
```

- ▶ This line calls the super class dispose method to make sure any resources used by the frame are properly freed.

```
System.exit(0);
```

► This line ensures that your program shuts down properly. The IBM Toolbox for Java connects to the iSeries server with user threads. Because of this, a failure to issue System.exit(0) may keep your Java program from properly shutting down.

## JDBCExampleDisplayAll class

This section examines the key methods of the JDBCExampleDisplayAll class.

### ***Instance variables***

The following instance variables for accessing the database are declared for the class:

```
private java.sql.Connection dbConnect;
private java.sql.PreparedStatement psAllRecord;
```

### ***The constructor method***

A non-default constructor is created for the class that takes parameters of a Connection object and a PreparedStatement object. This is done so that the main class (JDBCExample) can instantiate this class by passing the database objects already created (see Example 4-6).

---

#### *Example 4-6 Non-default constructor example*

---

```
public JDBCExampleDisplayAll(java.sql.Connection dbc, java.sql.PreparedStatement psAll)
{
    this();
    dbConnect = dbc;
    psAllRecord = psAll;
    this.setVisible(true);
}
```

---

#### **Class: JDBCExampleDisplayAll constructor**

Let's look more closely at the constructor:

```
this();
```

- This line executes the default constructor to take care of the window setup and initialization.

```
dbConnect = dbc;
psAllRecord = psAll;
```

- These lines set the instance variables for the database Connection and PreparedStatement to reference the objects passed from the JDBCExample class.

```
this.setVisible(true);
```

- This line displays the frame.

### ***The populateAllParts methods***

The method is called when the windowOpened event is invoked. The populateAllParts method runs the SQL statement to select all records from the parts file and populates the data to JTable components.

---

*Example 4-7 The populateAllParts method example*

---

```
public void populateAllParts() {  
  
    java.sql.ResultSet rs = null;  
    if (psAllRecord != null)  
    {  
        try  
        {  
            rs = psAllRecord.executeQuery();  
            while (rs.next())  
            {  
                String[] array = new String[5];  
                array[0] = rs.getString("PARTNO");  
                array[1] = rs.getString("PARTDS");  
                array[2] = insertSpaces(Integer.toString(rs.getInt("PARTQY")), 5);  
                array[3] = insertSpaces(rs.getBigDecimal("PARTPR").toString(), 8);  
                array[4] = rs.getDate("PARTDT").toString();  
                getDefaultTableModel1().addRow(array);  
            }  
        }  
        catch (Exception e)  
        {  
            showException(e);  
        }  
    }  
    return ;  
}
```

---

**Class:** JDBCExampleDisplayAll

**Method:** populateAllParts

Let's examine this method:

```
rs = psAllRecord.executeQuery();
```

- ▶ This line executes the SQL statement defined by the psAllRecord PreparedStatement object. It places the table of resulting records in a ResultSet object referenced by rs.

```
while (rs.next()) {
```

- ▶ The next() method of the ResultSet object attempts to position the cursor of the result set to the next record of the result table. The first time this happens, the cursor is pointed to the first record in the ResultSet object and returns true. If there are no records to retrieve, the method returns a false value. The method loops until the next() method returns a false value.

```
String[] array = new String[5];
```

- ▶ This line creates a new String array object, which receives the data from the current record of the ResultSet.

```
array[0] = rs.getString("PARTNO");  
array[1] = rs.getString("PARTDS");  
array[2] = insertSpaces(Integer.toString(rs.getInt("PARTQY")), 5);  
array[3] = insertSpaces(rs.getBigDecimal("PARTPR", 2).toString(), 8);  
array[4] = rs.getDate("PARTDT").toString();
```

- ▶ These lines retrieve each of the field values from the current record in the ResultSet. The values are converted to Strings and placed into the String array. Note the

`insertSpaces(String, int)` method, which is responsible for lining up numeric columns correctly.

```
getDefaultTableModel1().addRow(array);
```

- ▶ This line adds the string values of the current parts record as a new row to the default table model.

### 4.6.3 JDBC 2.0

The JDBC 2.0 API contains many new features, including scrollable result sets. The JDBC 2.0 API has been factored into two complementary components. The first component, which is termed the *JDBC 2.0 Core API*, comprises the updated contents of the `java.sql` package. The second component, termed the *JDBC 2.0 Optional Package API*, comprises the contents of a new package, `javax.sql`, which as its name implies will be delivered as a Java Standard Extension.

The `java.sql` package contains all of the enhancements that have been made to the existing JDBC interfaces and classes, in addition to a few new classes and interfaces. The new `javax.sql` package has been introduced to contain the parts of the JDBC 2.0 API that are closely related to other pieces of the Java platform. The parts are standard extensions, such as the Java Naming and Directory Interface (JNDI) and the Java Transaction Service (JTS). In addition, some advanced features that are easily separable from the JDBC Core API, such as connection pooling and rowsets, have also been added to `javax.sql`. Putting these advanced facilities into a standard extension, instead of into a core, helps keep the JDBC Core API small and focused.

Since the standard extensions are downloadable, it will always be possible to deploy an application that uses the features in the JDBC standard extension that will “run anywhere”. If a standard extension is not installed on a client machine, it can be downloaded along with the application that uses it.

The IBM Toolbox for Java Modification 4 supports both the JDBC 2.0 Core API and the JDBC 2.0 Optional Package API. If you want to use any of the following JDBC 2.0 enhancements, you also need to use Java 2 (JDK 1.2):

- ▶ BLOB interface
- ▶ CLOB interface
- ▶ Scrollable result sets
- ▶ Updatable result sets
- ▶ Batch update capability with Statement, PreparedStatement, and CallableStatement objects
- ▶ DataSource, AS400JDBCConnectionPool

To use BLOB and CLOB support with the iSeries server, the iSeries Universal Database (UDB) is required.

### JDBC result sets

A result set created by executing a statement may support the ability to move backward (last-to-first), as well as forward (first-to-last), through its contents. Result sets that support this capability are called *scrollable result sets*. Result sets that are scrollable also support relative and absolute positioning. *Absolute positioning* is the ability to move directly to a row by specifying its absolute position in the result set. *Relative positioning* gives the ability to move to a row by specifying a position that is relative to the current row. The definition of absolute and relative positioning in JDBC 2.0 is modeled on the X/Open SQL CLI specification.

The JDBC 1.0 API provided one result set type, forward-only. The JDBC 2.0 API provides three result set types:

- ▶ Forward-only
- ▶ Scroll-insensitive
- ▶ Scroll-sensitive

As their names suggest, the new result set types support scrolling, but they differ in their ability to make changes visible while they are open.

A scroll-insensitive result set is generally not sensitive to changes that are made while it is open. A scroll-insensitive result set provides a static view of the underlying data it contains. The membership, order, and column values of rows in a scroll-insensitive result set are typically fixed when the result set is created. On the other hand, a scroll-sensitive result set is sensitive to changes that are made while it is open. It also provides a “dynamic” view of the underlying data. For example, when using a scroll-sensitive result set, changes in the underlying column values of rows are visible. The membership and ordering of rows in the result set may be fixed. This is implementation defined.

An application may choose from two different concurrency types for a result set: read-only and updatable. A result set that uses read-only concurrency does not allow updates of its contents. This can increase the overall level of concurrency between transactions, since any number of read-only locks may be held on a data item simultaneously. A result set that is updatable allows updates and may use database write locks to mediate access to the same data item by different transactions. Since only a single write lock may be held at a time on a data item, this can reduce concurrency. Alternatively, an optimistic concurrency control scheme may be used if conflicting access to data will be rare. Optimistic concurrency control implementations typically compare rows either by value or by a version number to determine if an update conflict has occurred.

Two performance hints may be given to a JDBC 2.0 driver to make access to result set data more efficient. Specifically, the number of rows to be fetched from the database each time more rows are needed can be specified. A direction for processing the rows – forward, reverse, or unknown – can be given as well. These values can be changed for an individual result set at any time. A JDBC driver may ignore a performance hint if it chooses. The IBM Toolbox for Java JDBC 2.0 driver implements the number of rows fetched, but not the direction for processing.

## Using scrollable and updatable result sets

If a result set is created by executing a statement or prepared statement, you can move (scroll) backward (last-to-first) or forward (first-to-last) through the rows in a table.

Example 4-8 shows a code snippet that uses a scrollable result set.

---

### *Example 4-8 Scrollable result set example*

---

```
private java.sql.ResultSet rs;
private java.sql.PreparedStatement s;

s = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE PARTQY > ? ",
    java.sql.ResultSet.TYPE_SCROLL_INSENSITIVE,
    java.sql.ResultSet.CONCUR_READ_ONLY);
s.setFetchSize(25);
s.setInt(1,1000);
rs = s.executeQuery();
rs.next ();
.
.
.
rs.previous();
```

```
.  
rs.first();  
. . .  
rs.last();  
. . .  
boolean islast = rs.isLast();  
. . .  
boolean setPosition = rs.absolute(10);
```

---

The `prepareStatement` method now has two additional parameters. We set them to scroll insensitive and read only. We use the `setFetchSize` method to set the number of rows to be fetched from the database when more rows are needed. This may be changed at any time. If the value specified is zero, the driver will choose an appropriate fetch size.

There are a number of new methods that are available for moving through the result set. Some of them include:

- ▶ **Next**: Positions the cursor to the next row.
- ▶ **Previous**: Positions the cursor to the previous row.
- ▶ **First**: Positions the cursor to the first row of the result set.
- ▶ **Last**: Positions the cursor to the last row of the result set.
- ▶ **IsLast**: Indicates if the cursor is positioned on the last row.
- ▶ **Absolute**: Positions the cursor to an absolute row number. Attempting to move beyond the first row positions the cursor before the first row. Attempting to move beyond the last row positions the cursor after the last row. If the absolute row number is positive, this positions the cursor with respect to the beginning of the result set. If the absolute row number is negative, this positions the cursor with respect to the end of result set.

Example 4-9 shows a code snippet that updates a row from a result set.

---

*Example 4-9 Updatable result set example*

---

```
private java.sql.ResultSet rs;  
private java.sql.PreparedStatement s;  
  
s = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE PARTQY > ? FOR UPDATE",  
                           java.sql.ResultSet.TYPE_SCROLL_SENSITIVE,  
                           java.sql.ResultSet.CONCUR_UPDATABLE);  
  
s.setInt(1,1000);  
rs = s.executeQuery();  
rs.next ();  
. . .  
rs.previous();  
String name = rs.getString("PARTNO");  
int qty = rs.getInt("PARTQY");  
. . .  
. . . // application logic  
. . .  
rs.updateInt("PARTQY", qty); // Update the quantity with a new value  
  
rs.updateRow (); // Send the updates to the server.
```

---

In this case, we do not set a fetch size because the Toolbox JDBC driver will return only one row at a time from the iSeries server. The host server will lock the row on which we are positioned. We use the updateRow method to update the row.

## JDBC DataSource and connection pool

The DataSource interface provides an alternative to the DriverManager class for making a connection to a data source. Using a DataSource implementation is better for two important reasons:

- ▶ It makes code more portable.
- ▶ It makes code easier to maintain.

A DataSource object represents a real world data source. Depending on how it is implemented, the data source can be anything from a relational database to a spreadsheet or a file in tabular format.

Information about the data source and how to locate it, such as its name, the server on which it resides, its port number, and so on, is stored in the form of properties on the DataSource object. This makes an application more portable because it does not need to hard code a driver name, which often includes the name of a particular vendor, the way an application using the DriverManager class does.

It also makes maintaining the code easier, which can be explained in this example. If the data source is moved to a different server, all that needs to be done is to update the relevant property. None of the code using that data source needs to be touched.

Connection pooling is a mechanism, where when an application closes a connection, the connection is recycled rather than destroyed. Because establishing a connection is an expensive operation, reusing connections can improve performance dramatically by cutting down on the number of new connections that need to be created.

## Using DataSources and connection pools

IBM Toolbox for Java Modification 4 provides the AS400JDBCConnectionPoolDataSource class to create a factory for the creation of DataSource objects. It allows you to use the AS400JDBCConnectionPool class to represent a pool of iSeries JDBC connections that are available for a Java program. See Example 4-10.

---

### *Example 4-10 DataSource and connection pool example*

---

```
AS400JDBCConnectionPoolDataSource dataSource = new
AS400JDBCConnectionPoolDataSource(systemName, userid, password);

// Create a AS400JDBCConnectionPool.
AS400JDBCConnectionPool pool = new AS400JDBCConnectionPool(dataSource);

// Allow a maximum of 128 connections in the pool.
pool.setMaxConnections(128);

// Add 5 connections to the pool that can be used by the application.
pool.fill(5);

// Get a handle to a database connection from the pool.
dbConnect = pool.getConnection();
```

---

The `setMaxConnections()` method allows you to set the maximum connections that can be created in the pool. The `fill()` method is used to pre-create a number of live connections in the pool. You can use the `getConnection()` method to retrieve connections from the pool.

#### 4.6.4 JDBC 2.0 example

In this section, we create the `JDBC2OptionPackageExample` and `JDBC2OptionPackageExampleDisplayAll` classes based on the `JDBCExample` and `JDBCExampleDisplayAll` classes discussed in “`JDBCExample` class” on page 154. We use a data source, a connection pool, and scrollable result sets.

Example 4-11 shows the `connectToDB` method of the `JDBC2OptionPackageExample` class.

*Example 4-11 The connectToDB method*

---

```
public String connectToDB(String systemName, String userid, String password)
{
    try
    {
        setCursor(java.awt.Cursor.getPredefinedCursor(java.awt.Cursor.WAIT_CURSOR));
        AS400JDBCCollectionPoolDataSource dataSource = new
            AS400JDBCCollectionPoolDataSource(systemName, userid, password);
        AS400JDBCCollectionPool pool = new AS400JDBCCollectionPool(dataSource);
        pool.setMaxConnections(1);
        pool.fill(1);
        dbConnect = pool.getConnection();
        psSingleRecord = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE PARTNO = ?");

        psAllRecord = dbConnect.prepareStatement("SELECT * FROM PARTS ORDER BY PARTNO",
            java.sql.ResultSet.TYPE_SCROLL_SENSITIVE,
            java.sql.ResultSet.CONCUR_READ_ONLY);
        psAllRecord.setFetchSize(10);

        psUpdateRecord = dbConnect.prepareStatement("UPDATE PARTS SET PARTDS = ?, " + "
            PARTQY = ?, PARTPR = ?, PARTDT = ? WHERE PARTNO = ?");
        psAddRecord = dbConnect.prepareStatement("INSERT INTO PARTS (PARTDS, PARTQY, " + "
            PARTPR, PARTDT, PARTNO) VALUES (?, ?, ?, ?, ?)");
        psDeleteRecord = dbConnect.prepareStatement("DELETE FROM PARTS WHERE PARTNO = ?");

        }catch (Exception e){
            e.printStackTrace();
            showException(e);
            setCursor(java.awt.Cursor.getPredefinedCursor(java.awt.Cursor.DEFAULT_CURSOR));
            return "Connect Failed.";
        }
        setCursor(java.awt.Cursor.getPredefinedCursor(java.awt.Cursor.DEFAULT_CURSOR));
        return "Connected to AS/400.";
    }
}
```

---

We use the `AS400JDBCCollectionPoolDataSource` constructor to create a `DataSource` object named `dataSource`. The `AS400JDBCCollectionPool` constructor creates a new JDBC data source connection pool. The `setMaxConnections` method specifies the maximum number of connections that can be produced in the connection pool. The `fill` method makes the connections ready for applications to use. The application can use the `getConnection` method to get an unused connection from the connection pool.

In this example, we only use one connection. However, we can also develop multi-threaded server applications in which we use the connections from a thread. In this case, we set the maximum number of connections to a larger number, for example 128. In the threaded methods, we get a connection, use it, and then return it to the pool. This way, we can use multiple connections concurrently and still manage the connections through the `DataSource` object.

The `prepareStatement` method for the `psAllRecord` `PreparedStatement` object has two additional parameters. We set them to scroll sensitive and read only. *Scroll sensitive* means that any changes to the database will be reflected in the result set as we scroll through it. The `CONCUR_READ_ONLY` option allows us to only read rows from the database. We cannot update them. This setting allows the JDBC driver to function more efficiently because it can retrieve multiple rows at a time from the server. If we set this value to `Concur_Updatable`, the server returns only one row at a time. We use the `setFetchSize` method to set the number of rows to be fetched from the database (when more rows are needed) to ten. This option can be valuable when the SQL statement executed returns a very large result set. In JDBC 1.0, all the rows were returned to you. Now, you can control the number of rows returned programmatically. As you scroll through the result set, the driver retrieves the number of rows that you specify as they are needed.

Figure 4-8 shows the `JDBC2OptionPackageExampleDisplayAll` display. Three new buttons have been added to demonstrate some of the new JDBC 2.0 capabilities:

- ▶ Next 10
- ▶ Previous 10
- ▶ First 10

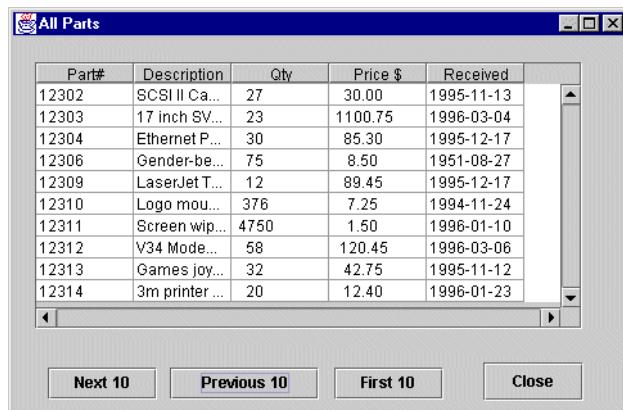


Figure 4-8 `JDBC2OptionPackageExampleDisplayAll`

Example 4-12 shows the `populateNextTen` method. This method supports the Next 10 button.

#### Example 4-12 The `populateNextTen` method

```
public void populateBoxNextTen() {
    int i = 0;
    try {
        if (!(psAllRecordRS.next())) /* read past the end of the result set*/
            psAllRecordRS.absolute(psAllRecordRS.getRow() - 1); /*return to last row */
        return;      /* Just leave current screen up */
    }

    removeAllRecord();
    do
```

```

{
    String[] array = new String[5];
    array[0] = psAllRecordRS.getString("PARTNO");
    array[1] = psAllRecordRS.getString("PARTDS");
    array[2] =
        insertSpaces(Integer.toString(psAllRecordRS.getInt("PARTQY")), 5);
    array[3] = insertSpaces(psAllRecordRS.getBigDecimal("PARTPR", 2).toString(), 8);
    array[4] = psAllRecordRS.getDate("PARTDT").toString();
    getDefaultTableModel1().addRow(array);
    i++;
}
}while ((i < 10) && (psAllRecordRS.next()));
}
catch (Exception e)
{
    showException(e);
}
return;
}

```

---

If we are already at the end of the result set, we use the absolute method to position to the last row and return. Otherwise, we use the next method to retrieve up to ten rows from the result set and add them to the JTable.

Example 4-13 shows the populatePrevTen method. This method supports the Previous 10 button.

*Example 4-13 The populatePrevTen method*

---

```

public void populatePrevTen() {
    int i = 0;

    try
    {
        psAllRecordRS.relative(-20);
        if(psAllRecordRS.getRow() == 0)
            psAllRecordRS.beforeFirst(); /* position before first */
        removeAllRecord();
        while ((i < 10) && psAllRecordRS.next())
        {
            String[] array = new String[5];
            array[0] = psAllRecordRS.getString("PARTNO");
            array[1] = psAllRecordRS.getString("PARTDS");
            array[2] =
                insertSpaces(Integer.toString(psAllRecordRS.getInt("PARTQY")), 5);
            array[3] =
                insertSpaces(psAllRecordRS.getBigDecimal("PARTPR", 2).toString(), 8);
            array[4] = psAllRecordRS.getDate("PARTDT").toString();
            getDefaultTableModel1().addRow(array);
            i++;
        }
    }
    catch (Exception e)
    {
        showException(e);
    }
    return;
}

```

---

We move the cursor backwards by 20 rows with the psAllRecordRS.relative(-20) statement. Notice that the getRow method returns the absolute row number. It returns zero if we are not positioned on a row. We can also point directly to a specific row with the absolute method. We can point the cursor to before the first row with the beforeFirst method.

Example 4-14 shows the populateFirstTen method. This method supports the First 10 button.

Example 4-14 The populateFirstTen method

---

```
public void populateFirstTen() {
    int i = 0;

    try
    {
        psAllRecordRS.first();
        removeAllRecord();
        do
        {
            String[] array = new String[5];
            array[0] = psAllRecordRS.getString("PARTNO");
            array[1] = psAllRecordRS.getString("PARTDS");
            array[2] = insertSpaces(Integer.toString(psAllRecordRS.getInt("PARTQY")), 5);
            array[3] = insertSpaces(psAllRecordRS.getBigDecimal("PARTPR", 2).toString(), 8);
            array[4] = psAllRecordRS.getDate("PARTDT").toString();
            getDefaultTableModel1().addRow(array);
            i++;
        }while ((i < 10) && (psAllRecordRS.next()));
    }
    catch (Exception e)
    {
        showException(e);
    }
    return;
}
```

---

Notice that the first() method positions the cursor to the first row of the result set. We then use the next() method to retrieve nine more rows from the result set and add them to the JTable.

#### 4.6.5 Reusable GUI part

For the remainder of the database examples in this chapter, a reusable class is used to handle the user interface. The advantage is that the user interface is designed, programmed, and tested once. Then, it is re-used in multiple applications that demonstrate different methods of accessing resources on the iSeries server.

The class is called *ToolboxGUI*. It is a subclass of javax.swing.JPanel and can be dropped onto a JFrame. ToolboxGUI communicates with its parent container through a PartsContainer interface. This interface allows specific methods of the parent to be invoked by the ToolboxGUI class to handle functions such as connecting to the database, retrieving, updating, or deleting a part record, or adding part records to a JTable.

To use the ToolboxGUI, we create a new class, which is a sub-class of javax.swing.JFrame, and implement the PartsContainer interface. In the Visual Composition Editor, we choose the ToolboxGUI bean and drop it on the empty application frame, re-size, and re-position to fit.

The PartsContainer interface designates methods to be implemented in the main application class so that ToolboxGUI can make requests for database access. The interface methods are:

- ▶ **connectToDB**: Connects to the database server and returns a String result.
- ▶ **getRecord**: Retrieves a single record from the database and places the resulting record field values in the JTextFields passed.
- ▶ **populateAllParts**: Retrieves all records from the database and adds values for each record in the JTable widget passed.
- ▶ **updateRecord**: Updates or adds the database record with the values passed; returns a String result.
- ▶ **deleteRecord**: Deletes a single record from the database; returns a String result.

ToolboxGUI calls out to the root parent method using the code in Example 4-15:

---

*Example 4-15 Finding the ToolboxGUI root container and calling its method*

---

```
rootParent = getParent();
while(rootParent.getParent() != null)
{
    rootParent = rootParent.getParent();
}

try
{
    ((PartsContainer)rootParent).connectToDB(systemName, userid, password);
}catch (Exception e)
{...}
```

---

Let's examine the code above:

```
rootParent = getParent();
while(rootParent.getParent() != null)
{
    rootParent = rootParent.getParent();
}
```

- ▶ These lines execute to find out the ToolboxGUI's root container, which implements the listed PartsContainer interface methods.
- 
- ```
((PartsContainer)rootParent).connectToDB(systemName, userid, password);
```
- ▶ This line casts the rootParent as an object that conforms to the PartsContainer interface. The connectToDB method is invoked on the parent object. Similar code is used for the other interface methods.

The ToolboxGUI class also has a helper class, DisplayAllParts, to display and populate data in the JTable. This class is instantiated when the Get All Parts button is clicked. It uses the same mechanism defined previously to call out to the parent's populateAllParts method.

#### 4.6.6 Stored procedures

Using stored procedures is an extension of the JDBC access technique. Instead of using PreparedStatement and Statement objects to execute SQL statements, a CallableStatement object is defined and executed.

The `prepareCall` method on the `Connection` object is used to create a `CallableStatement` object, for example:

```
CallableStatement aCS = aConnection.prepareCall(  
    "CALL LibraryName.PocedureName(?, ?, ?)");
```

These lines define a `CallableStatement` object, `aCS`. When executed, `aCS` calls the procedure in the specified library and passes three parameters. These parameters can be input, output, or both. Output parameters must be registered using the `registerOutParameter` method, for example:

```
aCS.registerOutParameter (3, java.sql.Types.INTEGER);
```

This line registers the third parameter (the third question mark) as an output parameter for the stored procedure of an SQL type integer. After the procedure is run, the value of the parameter can be retrieved using `aCS.getInt(3)`. Other *getters* (= methodname starting with `get`) exist for each registered data type.

Input parameters must be set using the `set` method associated with the data type, for example:

```
aCS.setInt(1, 500);
```

This line sets the value of the first parameter to an integer value of 500.

Stored procedures can be executed using the `execute`, `executeQuery`, or `executeUpdate` methods. The `execute` method is used when zero or more result sets are expected to be returned. The `executeQuery` method can be used when exactly one result set is returned. The `executeUpdate` can be used when no result set but a number of rows affected is to be returned for database operations such as `UPDATE`, `INSERT`, or `DELETE`.

Stored procedures are generally used for two reasons. First, native programs written in RPG, COBOL, and others can be used by the Java application through a standard interface. Second, stored procedures can greatly boost performance of the application when compared with straight SQL.

#### 4.6.7 JDBC stored procedure application example

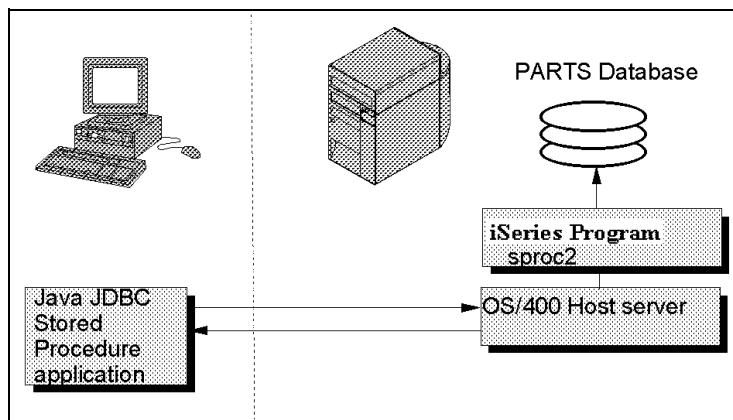


Figure 4-9 JDBC application stored procedures

In the following example, JDBC stored procedures are used to access records in an iSeries database (see Figure 4-9).

The client program requests data from the iSeries database by calling an iSeries stored procedure program. The host server passes the call to the iSeries program and returns the results to the client program in an SQL result set. The JDBC support

handles all data conversions. Figure 4-10 and Figure 4-11 show the user interface to the stored procedure.

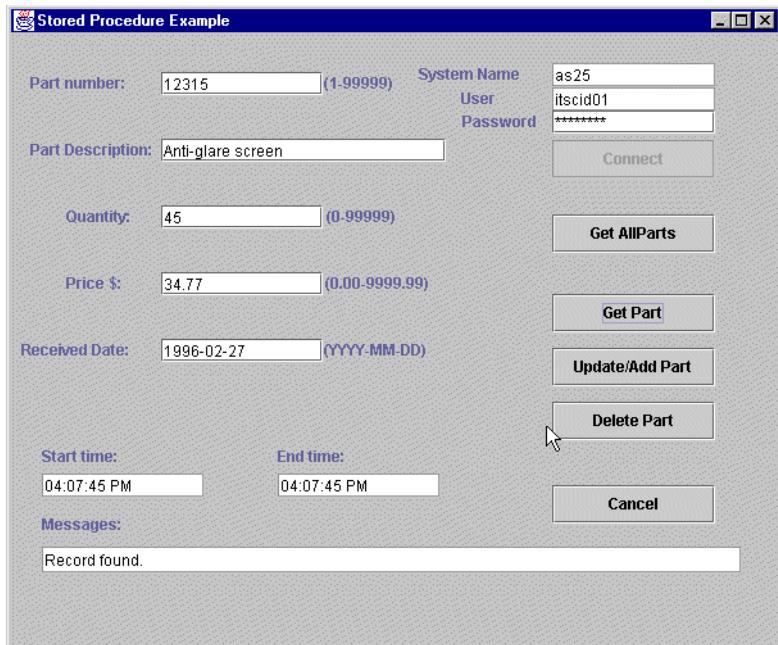


Figure 4-10 Stored procedure example: A single part

| All Parts |                 |      |          |            |  |
|-----------|-----------------|------|----------|------------|--|
| Part#     | Description     | Qty  | Price \$ | Received   |  |
| 12302     | SCSI II Ca...   | 27   | 30.00    | 1995-11-13 |  |
| 12303     | 17 inch SV...   | 23   | 1100.75  | 1996-03-04 |  |
| 12304     | Ethernet P...   | 30   | 85.30    | 1995-12-17 |  |
| 12306     | Gender-be...    | 75   | 8.50     | 1991-08-27 |  |
| 12309     | LaserJet T...   | 12   | 89.45    | 1995-12-17 |  |
| 12310     | Logo mou...     | 376  | 7.25     | 1994-11-24 |  |
| 12311     | Screen wip...   | 4750 | 1.50     | 1996-01-10 |  |
| 12312     | V34 Mode...     | 58   | 120.45   | 1996-03-06 |  |
| 12313     | Games joy...    | 32   | 42.75    | 1995-11-12 |  |
| 12314     | 3m printer ...  | 20   | 12.40    | 1996-01-23 |  |
| 12315     | Anti-glare s... | 45   | 34.77    | 1996-02-27 |  |
| 12316     | Quad spee...    | 14   | 151.38   | 1996-01-12 |  |
| 12317     | SCSI II Ca...   | 25   | 37.84    | 1995-11-13 |  |
| 12318     | 17 inch SV...   | 6    | 1388.59  | 1996-03-04 |  |
| 12319     | Ethernet P...   | 30   | 107.60   | 1995-12-17 |  |
| 12320     | Home mou...     | 47   | 32.16    | 1996-02-18 |  |
| 12321     | Gender-be...    | 75   | 10.71    | 1991-08-27 |  |
| 12322     | 600 dpi flat... | 12   | 1104.21  | 1996-03-01 |  |
| 12323     | 100 MHZ P...    | 4    | 2365.55  | 1996-02-24 |  |
| 12324     | LaserJet T...   | 12   | 112.83   | 1995-12-17 |  |

Figure 4-11 Stored procedure example: All parts

Class `StoredProcedureExample` is the main class in this application. It is functionally equivalent to the `JDBCExample` application, but is implemented using different techniques. The `ToolboxGUI` class is used to handle all user interaction. A stored procedure is used instead of SQL statements. Record read, update, add, and delete are implemented in this example. This was done by creating two stored procedures that perform the corresponding database operations. The reason for creating two stored procedures was due to the different number of parameters needed for the different database operations.

The programs we use as stored procedures are written in RPG and named SPROC2 and SPROC3 in library APILIB. The first program SPROC2 takes two integer input parameters. Parameter 1 is an action code. A value of 1 returns a single record in the result set with the part number field matching the part number supplied in the second parameter. A value of 2 in

the first parameter returns all records from the parts database in a result set. The second parameter is ignored in this case. A value of 3 deletes a single record in the database file with the part number field matching the part number supplied in the second parameter. The second program SPROC3 takes three integers, one string, one float, and one date input parameter. Parameter 1 is an action code. A value of 1 causes a single record defined by the part number supplied in the second parameter to be updated. Parameters three through six supply the values for the corresponding database fields. A value of 2 causes a single record defined by the part number supplied in the second parameter to be written into the database file. Parameters three through six supply the values for the corresponding database fields.

## **StoredProcedureExample class**

This section investigates the key methods of the `StoredProcedureExample` class. SQL statements are written in capital letters for better readability. They can also be written in lowercase.

### ***Instance variables***

The following instance variables for accessing the database are declared for the class:

```
private java.sql.CallableStatement callableStmt;
private java.sql.CallableStatement callableStmt1;
private java.sql.Connection dbConnect;
private ToolboxGUI ivjToolboxGUI1 = null;
```

### ***The connectToDB method***

The `connectToDB` method is called by the `ToolboxGUI` class when the Connect button is clicked. String parameters representing the iSeries server name, user ID, and password are passed to the method.

---

#### *Example 4-16 Stored procedure example connectToDB method*

---

```
public void connectToDB(String systemName, String userid, String password) throws Exception {
    java.sql.DriverManager.registerDriver(new com.ibm.as400.access.AS400JDBCDriver());
    dbConnect = java.sql.DriverManager.getConnection("jdbc:as400://" + systemName +
        "/apilib;naming=sql;errors=full;date format=iso;extended dynamic=true;" +
        "package=TeamLab;package library=apilib", userid, password);
    try
    {
        dbConnect.createStatement().execute("DROP PROCEDURE APILIB.PARTQRY2");
    }
    catch (Exception e){}
    try
    {
        dbConnect.createStatement().execute("CREATE PROCEDURE APILIB.PARTQRY2(IN P1" +
            " INT, IN P2 INT) RESULT SETS 1 LANGUAGE RPG DETERMINISTIC CONTAINS SQL" +
            " EXTERNAL NAME APILIB.SPROC2 PARAMETER STYLE GENERAL");
    }
    catch (Exception e){}
    callableStmt = dbConnect.prepareCall("CALL APILIB.PARTQRY2(?, ?)");
    try
    {
        dbConnect.createStatement().execute("DROP PROCEDURE APILIB.PARTQRY3");
    }
    catch (Exception e){}
    try
    {
        dbConnect.createStatement().execute("CREATE PROCEDURE APILIB.PARTQRY3(IN P1" +
            " INT, IN P2 INT , IN P3 CHAR (25), IN P4 INT , IN P5 DEC (6, 2), IN P6 DATE)" +
```

```

    " LANGUAGE RPG NOT DETERMINISTIC CONTAINS SQL EXTERNAL NAME APILIB.SPROC3" +
    " PARAMETER STYLE GENERAL");
}
catch (Exception e){}
callableStmt1 = dbConnect.prepareCall("CALL APILIB.PARTQRY3(?, ?, ?, ?, ?, ?)");
return;
}

```

---

Let's take a closer look at this method:

```

java.sql.DriverManager.registerDriver(new
com.ibm.as400.access.AS400JDBCDriver());
dbConnect = java.sql.DriverManager.getConnection("jdbc:as400://" + systemName
+ "/apilib;naming=sql;errors=full;date format=iso;extended dynamic=true;" +
"package=TeamLab;package library=apilib", userid, password);
▶ Loads the JDBC driver and connects to the iSeries server the same way as in the
preceding JDBCExample class.

dbConnect.createStatement().execute("DROP PROCEDURE APILIB.PARTQRY2");
▶ Attempts to remove the stored procedure PARTQRY2 from the system catalog, if it exists.
If the procedure does not already exist in the catalog, an error is thrown, so we catch it and
do nothing.

dbConnect.createStatement().execute("CREATE PROCEDURE APILIB.PARTQRY2(IN P1" + " INT, IN P2
INT) RESULT SETS 1 LANGUAGE RPG DETERMINISTIC CONTAINS SQL" +" EXTERNAL NAME APILIB.SPROC2
PARAMETER STYLE GENERAL");
▶ This command executes an SQL statement to add the PARTQRY2 procedure to the
system catalog. A new statement object is created by the Connection object. The execute
method is used to run an ad hoc SQL statement to declare the RPG program SPROC2 to
the catalog. In a production environment, the procedure is added to the catalog once by a
system administrator and not added on the fly by an application each time it connects to
the database.

```

**Attention:** We show how to drop and create an iSeries stored procedure from a Java client here. In most cases, it is better to do this directly on the iSeries server. You can do this on the iSeries server by using interactive SQL, Operations Navigator, or an application program. Creating the stored procedure needs to be done only once. It is added to the system catalog, so it can be found and reused. Creating a stored procedure from the client, as shown here, adds extra overhead to a Java application.

```

callableStmt = dbConnect.prepareCall("CALL APILIB.PARTQRY2(?, ?)");
▶ This command creates a new CallableStatement object from the Connection object. The
statement declares the stored procedure and has markers for two parameters. The
parameters are input only, because no output parameters are registered.

```

### ***The getRecord method***

The GetRecord method is called by the ToolboxGUI class when the Get Part button is clicked.

---

*Example 4-17 Stored procedure example getRecord method*

---

```
public String getRecord(String partNo, javax.swing.JTextField partDesc,
javax.swing.JTextField partQty, javax.swing.JTextField partPrice, javax.swing.JTextField
partDate) throws Exception
{
    java.sql.ResultSet rs = null;
    callableStmt.setInt(1, 1);
    callableStmt.setInt(2, Integer.parseInt(partNo));
    rs = callableStmt.executeQuery();
    if (rs.next())
    {
        partDesc.setText(rs.getString(2).trim());
        partQty.setText(Integer.toString(rs.getInt(3)));
        partPrice.setText(rs.getBigDecimal(4).setScale(2).toString());
        partDate.setText(rs.getDate(5).toString());
    }
    else
    {
        partDesc.setText("");
        partQty.setText("0");
        partPrice.setText("0.00");
        partDate.setText("");
        return "Record not found.";
    }
    return "Record found.";
}
```

---

**Class:** StoredProcedureExample

**Method:** getRecord

The method highlights are listed here:

```
java.sql.ResultSet rs = null;
```

- ▶ Declares a variable, rs, to reference a ResultSet object.

```
callableStmt.setInt(1, 1);
callableStmt.setInt(2, Integer.parseInt(partNo.trim()));
```

- ▶ Uses the setInt method to set the value of parameter 1 to the integer value 1 to tell the program to get a single part record. Then, the setInt method is used to set the value of parameter 2 to the integer value of the part number passed on the parameter list.

```
rs = callableStmt.executeQuery();
```

- ▶ Executes the stored procedure defined by the callableStatement object and places the table of resulting records in a ResultSet object referenced by rs.

```
if (rs.next()) {
```

- ▶ The next() method of the ResultSet attempts to position the cursor of the result set to the next record from the result table. Because this is the first read from the result set, the method positions to the first record from the result set and returns a true value. If there are no records to retrieve, the method returns a false value.

```
partDesc.setText(rs.getString(2).trim());
partQty.setText(Integer.toString(rs.getInt(3)));
partPrice.setText(rs.getBigDecimal(4).setScale(2).toString());
partDate.setText(rs.getDate(5).toString());
```

- ▶ These lines retrieve values of database fields and place them in their corresponding screen fields. The ResultSet object has getter methods for many Java data types. We use column indices instead of column names to reference the values requested from the result set.

### **The populateAllParts method**

This method is called from the DisplayAllParts non-default constructor through the PartsContainer interface. It runs the SQL statement to select all records from the parts file (Example 4-18).

*Example 4-18 Stored procedure example populateAllParts method*

---

```
public void populateAllParts(javax.swing.table.DefaultTableModel defaultTableModel) throws Exception
{
    java.sql.ResultSet rs = null;
    callableStmt.setInt(1, 2);
    callableStmt.setInt(2, 0);
    rs = callableStmt.executeQuery();
    while (rs.next())
    {
        String[] array = new String[5];
        array[0] = rs.getString(1);
        array[1] = rs.getString(2);
        array[2] = DisplayAllParts.insertSpaces(Integer.toString(rs.getInt(3)), 5);
        array[3]=DisplayAllParts.insertSpaces((rs.getBigDecimal(4).setScale(2).toString()),8);
        array[4] = rs.getDate(5).toString();
        defaultTableModel.addRow(array);
    }
    return;
}
```

---

**Class:** StoredProcedureExample

**Method:** populateAllParts

The populateAllParts method highlights are listed here:

```
callableStmt.setInt(1, 2);
callableStmt.setInt(2, 0);
```

- ▶ Uses the setInt method to set the value of parameter 1 to the integer value 2 to tell the program to get all part records. Then, the setInt method is used to set the value of parameter 2 to the integer value of 0 so that a null value is not passed to the procedure.

```
rs = callableStmt.executeQuery();
```

- ▶ Executes the stored procedure defined by the CallableStatement object, and places the table of resulting records in a ResultSet object referenced by rs.

```
while (rs.next()) {
```

- ▶ The next() method of the ResultSet attempts to position the cursor of the result set to the next record from the result table. The first time the cursor is pointed to the first record in ResultSet and returns a true value. If there are no records to retrieve, the method returns a false value. The method loops until next() returns a false value.

```

array[0] = rs.getString(1);
array[1] = rs.getString(2);
array[2] = DisplayAllParts.insertSpaces(Integer.toString(rs.getInt(3)),5);
array[3] = DisplayAllParts.insertSpaces((rs.getBigDecimal(4).setScale(2).toString()),8);
array[4] = rs.getDate(5).toString();

► Retrieves the field values from the current record in the ResultSet. These values are converted to strings and placed into a string array for adding to the defaultTableModel to display in the JTable.

defaultTableModel.addRow(array);

► Adds the String values of the current parts record as a new row at the end of the JTable.

```

### **The dispose method**

The dispose method is called when the application window is closed (Example 4-19).

---

#### *Example 4-19 Stored procedure example dispose method*

---

```

public void dispose()
{
    try
    {
        if (callableStmt != null)
        {
            callableStmt.close();
        }
        if (callableStmt1 != null)
        {
            callableStmt1.close();
        }
        if (dbConnect != null)
        {
            dbConnect.close();
        }
    }
    catch (Exception e)
    {
        System.out.println("Exception while disconnecting from AS/400." + e.toString());
    }
    super.dispose();
    System.exit(0);
    return;
}

```

---

**Class:** StoredProcedureExample

**Method:** dispose

The dispose method highlights include:

```
callableStmt.close();
```

- Releases the CallableStatements database and JDBC resources immediately. This also closes the current ResultSet. The CallableStatements should be tested for null, since we do not know whether the user has connected to the database. Closing a CallableStatement, which is null, throws an exception.

```
dbConnect.close();
```

- ▶ Disconnects from the iSeries server.

```
super.dispose();
```

- ▶ Calls the super class dispose method to make sure any resources used by the frame are properly freed.

```
System.exit(0);
```

- ▶ This line ensures that your program shuts down properly. IBM Toolbox for Java connects to the iSeries with user threads. Because of this, a failure to issue System.exit(0) may keep your Java program from properly shutting down.

#### 4.6.8 DDM record-level access application example

In the example in Figure 4-12, distributed data management (DDM) record-level access is used to access records in an iSeries database named Parts.

The client program requests data from the iSeries database by interfacing with the host DDM server. The DDM server accesses the database and returns the results to the client program.

We demonstrate using the DDM server to retrieve the format of the Parts file from the iSeries server. This makes it easy to work with the file using field names.

The RLAExample class is the main class in this application. We use the same classes as in the other examples, ToolboxGUI and DisplayAllParts, to handle all user interaction. See Figure 4-13 on page 180.

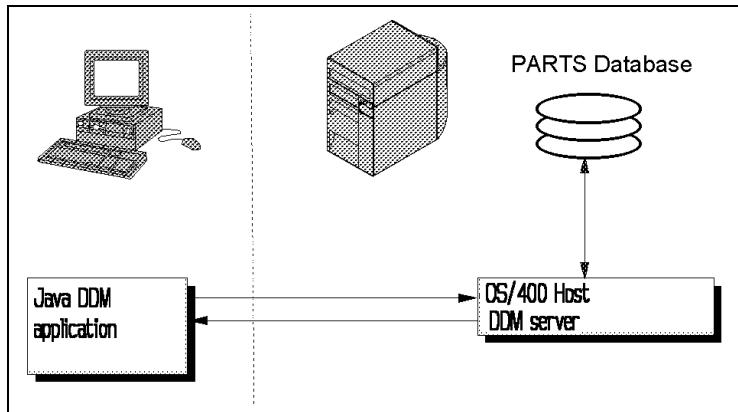


Figure 4-12 DDM record-level access

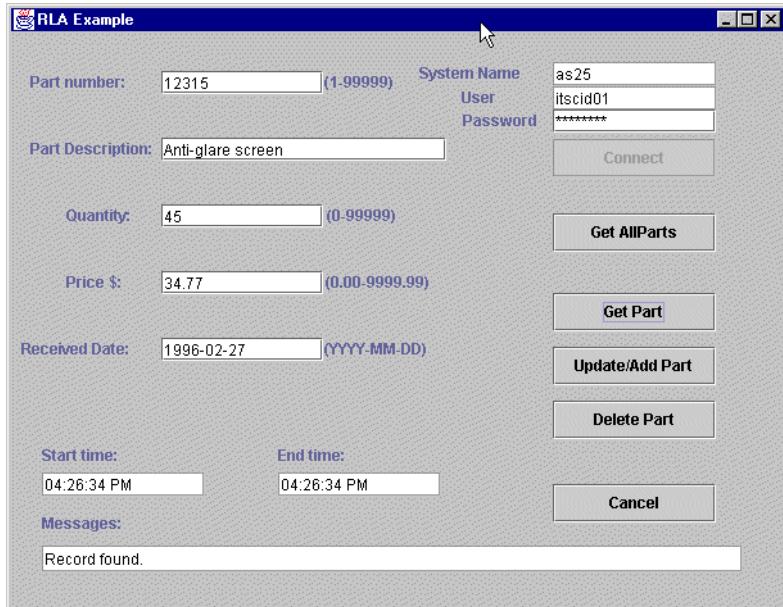


Figure 4-13 DDM record-level access example

## RLAExample class

This section looks at the key methods of the RLAExample class.

### Instance variables

The following instance variables for accessing the database are declared for the class:

```
private AS400 as400;
private KeyedFile myKeyedFile;
private SequentialFile mySeqFile;
private RecordFormat partsFormat = null;
private ToolboxGUI ivjToolboxGUI1 = null;
```

### The connectToDB method

The ConnectToDB method is called by the ToolboxGUI class when the Connect button is clicked.

---

#### Example 4-20 Record-level access example connectToDB method

```
public void connectToDB(String systemName, String userid, String password) throws Exception
{
    as400 = new AS400(systemName, userid, password);
    QSYSObjectPathName fileName = new QSYSObjectPathName("APILIB", "PARTS", "*FILE", "MBR");
    myKeyedFile = new KeyedFile(as400, fileName.getPath());
    mySeqFile = new SequentialFile(as400, fileName.getPath());
    try
    {
        as400.connectService(AS400.RECORDACCESS);
    }
    catch (Exception e)
    {
        System.out.println("Unable to connect");
        System.exit(0);
    }
    try
    {
```

```

AS400FileRecordDescription recordDescription = new AS400FileRecordDescription(as400,
"/QSYS.LIB/APILIB.LIB/PARTS.FILE");
partsFormat = recordDescription.retrieveRecordFormat()[0];
partsFormat.addKeyFieldDescription("PARTNO");
}
catch (Exception e)
{
    System.out.println("Unable to retrieve record format from APILIB/PARTS");
    System.exit(0);
}
try
{
    myKeyedFile.setRecordFormat(partsFormat);
    mySeqFile.setRecordFormat(partsFormat);
    myKeyedFile.open(AS400File.READ_WRITE, 0, AS400File.COMMIT_LOCK_LEVEL_NONE);
    mySeqFile.open(AS400File.READ_ONLY, 100, AS400File.COMMIT_LOCK_LEVEL_NONE);
}
catch (Exception e)
{
    System.out.println("Unable to open file");
    System.exit(0);
}
return;
}

```

---

Let's take a closer look at this method:

```
as400 = new AS400(systemName, userid, password);
```

- ▶ Creates a new iSeries connection object. System name, user ID, and password are passed through the constructor.

```
QSYSObjectPathName fileName = new QSYSObjectPathName("APILIB", "PARTS",
"*FILE", "MBR");
```

- ▶ Creates a new path name object for the file to be accessed.

```
myKeyedFile = new KeyedFile(as400, fileName.getPath());
mySeqFile = new SequentialFile(as400, fileName.getPath());
```

- ▶ Creates a keyed file object and a sequential file object that represent the file we access on the iSeries server. We use the QSYSObjectPathName object to get the path and name of the file into the correct format. The keyed file is used for all single-record operations such as add, update, delete, and read. The sequential file is used for the multiple-record operation in the populateAllParts method.

```
as400.connectService(AS400.RECORDACCESS);
```

- ▶ Connects to the iSeries DDM server. This is not required. If a service connection is needed and does not already exist, the service is connected automatically. We choose to place the connection overhead in the connect method as opposed to connecting the first time the user requests a part record.

```
AS400FileRecordDescription recordDescription = new
AS400FileRecordDescription(as400, "/QSYS.LIB/APILIB.LIB/PARTS.FILE");
```

- ▶ Creates the Record Description object for accessing the file. The record description is the same as the record format for file APILIB/PARTS.

```

partsFormat = recordDescription.retrieveRecordFormat()[0];
► We retrieve the record format. There is only one record format for the file, so we use the
first (and only) element of the RecordFormat array returned as the record format for the
file.

partsFormat.addKeyFieldDescription("PARTNO");
► We make the PARTNO field the key field.

myKeyedFile.setRecordFormat(partsFormat);
mySeqFile.setRecordFormat(partsFormat);
► We set the record format with the format description that we retrieved from the iSeries
server.

myKeyedFile.open(AS400File.READ_WRITE, 0, AS400File.COMMIT_LOCK_LEVEL_NONE);
mySeqFile.open(AS400File.READ_ONLY, 100, AS400File.COMMIT_LOCK_LEVEL_NONE);
► We open the keyed file for both read and write. Since we are not opening for read only, a
blocking factor (0) is ignored and no blocking is done. If we are reading records only, as in
the sequential file, we can specify a blocking factor (100) on the open to help achieve
better performance. We are not using commitment control.

```

### ***The getRecord method***

The getRecord method is called by the ToolboxGUI class when the Get Part button is clicked.

---

#### *Example 4-21 Record-level access example getRecord method*

---

```

public String getRecord(String partNo, javax.swing.JTextField partDesc,
javax.swing.JTextField partQty,javax.swing.JTextField partPrice, javax.swing.JTextField
partDate) throws Exception
{
    Object[] theKey = new Object[1];
    theKey[0] = new java.math.BigDecimal(partNo.trim());
    Record data = myKeyedFile.read(theKey);
    if (data != null)
    {
        partDesc.setText(((String) data.getField("PARTDS")).trim());
        partQty.setText(((java.math.BigDecimal) data.getField("PARTQY")).toString());
        partPrice.setText(((java.math.BigDecimal) data.getField("PARTPR")).toString());
        partDate.setText((String) data.getField("PARTDT"));
        return "Record found.";
    }
    else
    {
        partDesc.setText("");
        partQty.setText("0");
        partPrice.setText("0.00");
        partDate.setText("");
        return "Record not found.";
    }
}

```

---

The getRecord method highlights include:

```

Object[] theKey = new Object[1];
theKey[0] = new java.math.BigDecimal(partNo.trim());

```

- ▶ Creates the key for reading the records. The key for a keyed file is specified as an object array.

```
Record data = myKeyedFile.read(theKey);
```

- ▶ Reads the first record matching the key. Null is returned if the record is not found.

```
partDesc.setText(((String) data.getField("PARTDS")).trim());
partQty.setText(((java.math.BigDecimal) data.getField("PARTQY")).toString());
partPrice.setText(((java.math.BigDecimal)
data.getField("PARTPR")).toString());
partDate.setText((String) data.getField("PARTDT"));
return "Record found.;"
```

- ▶ If the record is found, we use the field names to retrieve the data and set the text property of the objects shown on the ToolboxGUI. This causes the text fields to display the values received in this method.

### ***The populateAllParts method***

The populateAllParts method is called from the DisplayAllParts class through the PartsContainer Interface. It uses the sequential file and requests all records from the parts file (see Example 4-22).

---

#### *Example 4-22 Record-level access example populateAllParts method*

---

```
public void populateAllParts(javax.swing.table.DefaultTableModel defaultTableModel) throws
Exception
{
    try
    {
        Record record = mySeqFile.readFirst();
        while (record != null)
        {
            String[] array = new String[5];
            array[0] = ((java.math.BigDecimal) record.getField("PARTNO")).toString();
            array[1] = (String) record.getField("PARTDS");
            array[2] = DisplayAllParts.insertSpaces(((java.math.BigDecimal)
                record.getField("PARTQY")).toString(), 5);
            array[3] = DisplayAllParts.insertSpaces(((java.math.BigDecimal)
                record.getField("PARTPR")).toString(), 8);
            array[4] = (String) record.getField("PARTDT");
            defaultTableModel.addRow(array);
            record = mySeqFile.readNext();
        }
    }
    catch (Exception e)
    {
        System.out.println("unable to get all");
        System.exit(0);
    }
    return;
}
```

---

The highlights of the populateAllParts method are listed here:

```
Record record = mySeqFile.readFirst();
```

- ▶ We use the readFirst method to read the first record.

```

array[0] = ((java.math.BigDecimal) record.getField("PARTNO")).toString();
array[1] = (String) record.getField("PARTDS");
array[2] = DisplayAllParts.insertSpaces(((java.math.BigDecimal)
    record.getField("PARTQY")).toString(), 5);
array[3] = DisplayAllParts.insertSpaces(((java.math.BigDecimal)
    record.getField("PARTPR")).toString(), 8);
array[4] = (String) record.getField("PARTDT");

```

- ▶ We use the field names to retrieve the data fields and move them to the array elements.

```
defaultTableModel.addRow(array);
```

- ▶ We use the addRow method to add a new row to the JTable.

```
record = mySeqFile.readNext();
```

- ▶ We read the next record in the file.

**Note:** When using a sequential file, the records are not sorted by the key defined in the database file. Use a keyed file if you want to obtain sorted records.

### **The dispose method**

The dispose method is called when the application window is closed. See Example 4-23.

---

#### Example 4-23 Record-level access example dispose method

---

```

public void dispose()
{
    try
    {
        if (myKeyedFile != null)
        {
            myKeyedFile.close();
        }
        if (mySeqFile != null)
        {
            mySeqFile.close();
        }
        if (as400 != null)
        {
            as400.disconnectAllServices();
        }
    }
    catch (Exception e)
    {
        System.out.println("Exception while disconnecting from AS/400." + e.toString());
    }
    super.dispose();
    System.exit(0);
    return;
}

```

---

**Class:** RLAExample

**Method:** dispose

The highlights of the dispose method are:

```

myKeyedFile.close();
▶ Closes the open database file.

as400.disconnectAllServices();
▶ Releases all connections to the iSeries server and releases resources associated with
server jobs processing requests for the client.

super.dispose();
▶ Calls the super class dispose method to make sure any resources used by the frame are
properly freed.

```

All other methods used in the dispose method are the same as in the JDBC and StoredProcedure examples.

#### 4.6.9 Distributed Program Call feature

The Distributed Program Call feature of the Toolbox allows a Java program to directly execute any non-interactive program object (\*PGM) on the iSeries server. It passes input data as parameters and returns results through output parameters.

The Java developer must use the data conversion classes from the Toolbox to convert input parameters from the Java format to an iSeries data type and convert output parameters from iSeries format to a Java format.

The advantage of using the ProgramCall class is that native iSeries non-interactive programs can be executed from a Java application unchanged. Native program calls can also result in better performance of a Java application when compared with JDBC. In addition, this interface can call programs on the iSeries server that do more than just database access. For example, a Java application can call a program that starts nightly job processing, saves libraries to tape, or sends or receives data through communication lines.

Calling a native iSeries program involves the following steps:

1. Connect to the iSeries server by creating an AS400 object.
2. Create a ProgramCall object.
3. Define and initialize a ProgramParameter array for passing parameters to and from the called program.
4. Use the Data Conversion classes to convert input parameter values from the Java format to the iSeries format.
5. Use the setProgram method to specify the qualified name of the program to call and the parameters to use, if not declared on the ProgramCall constructor.
6. Execute the program using the run method.
7. If the run method fails, obtain detailed error information through AS400Message objects.
8. Retrieve output parameters using the getOutputData method of the ProgramParameter object.
9. Convert output parameter values using the data conversion classes.

#### 4.6.10 Distributed Program Call (DPC) application example

In this example, we use the Distributed Program Call (DPC) interface to allow a client program to call an iSeries program (see Figure 4-14). We also develop this application using the Enterprise Toolkit/400 Program Call SmartGuide. See 5.3, "Distributed Program Call SmartGuide" on page 259, for details.

The client program requests data from the iSeries database by calling an iSeries program. Information is passed between the programs using parameters. It is up to the application implementer to handle data conversions. See Figure 4-15.

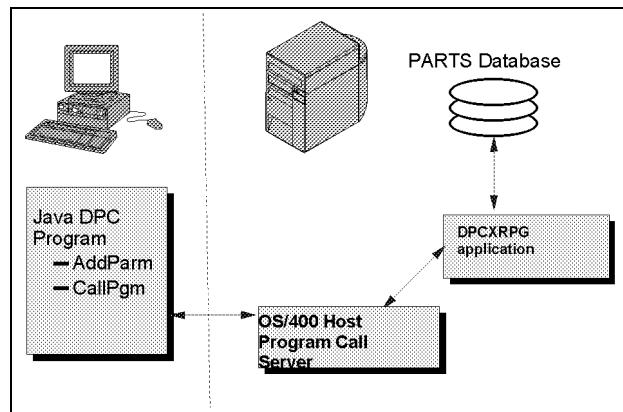


Figure 4-14 Distributed program call example flow

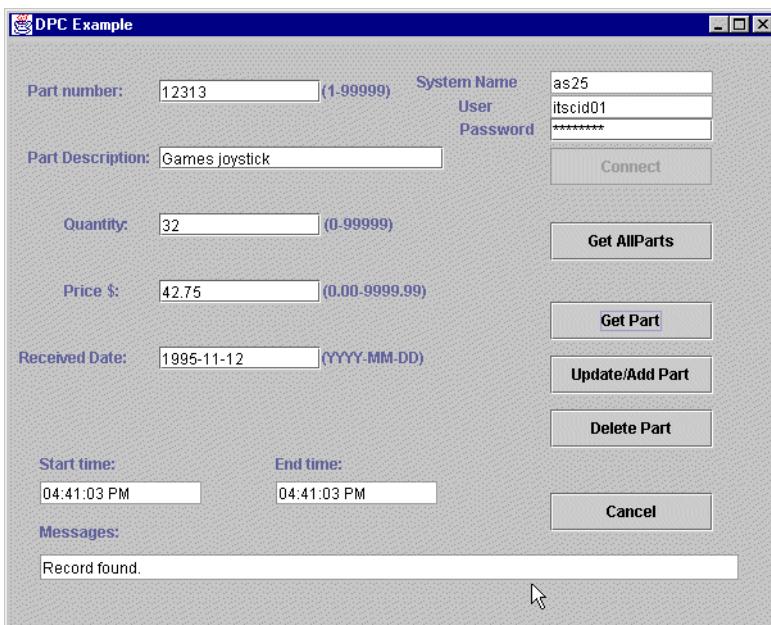


Figure 4-15 Distributed program call example

The client program requests data from the server program by calling it and passing it parameters. The input parameters are a flag and a part number and all of the attributes of a part. For example, S12313 is a request for a single record (Flag = S) of part number 12301. If requesting all parts (Flag=A), the part number is not necessary. The server program, DPCXRPG, searches the database for the requested information. The result is passed back in the output parameters.

The DPCEExample class is the main class in this application. It is functionally equivalent to the StoredProcedureExample application, but is implemented using different techniques. The ToolboxGUI and DisplayAllParts class are used to handle all user interaction. A native RPG program is called on the iSeries server to access and return data. All of the four basic database operations are implemented in this example.

## RPG program background

Library: APILIB

Program name: DPCXRPG

Parameters (all are used as Input/Output): Table 4-20

Table 4-20 Parameter list

| Sequence/field | Description                           | Length/type  |
|----------------|---------------------------------------|--------------|
| 1 / OPTION     | In: Operation Code / Out: Return Code | 1 character  |
| 2 / PARTNO     | Part Number                           | 5.0 packed   |
| 3 / PARTDS     | Part Description                      | 25 character |
| 4 / PARTQY     | Part Quantity                         | 5.0 packed   |
| 5 / PARTPR     | Part Price                            | 6.2 packed   |
| 6 / PARTDT     | Part Date Received                    | 10 date      |

Values of the Operation Code (Input OPTION): Table 4-21

Table 4-21 Flag operation code

| Operation code | Database operation to execute                                                                                                                                     |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| S              | Retrieve a single record for the supplied key.                                                                                                                    |
| A              | Retrieve all records.                                                                                                                                             |
| F              | Fetch next record based on the current position.                                                                                                                  |
| E              | End the program.                                                                                                                                                  |
| D              | Delete a single record for the supplied key.                                                                                                                      |
| U              | Update a single record for the supplied key with the attribute data. Write a single record for the supplied key with the attribute data if it does not yet exist. |

Values of the Return Code (Output OPTION): Table 4-22

Table 4-22 Flag operation codes

| Return code | Result description                                                          |
|-------------|-----------------------------------------------------------------------------|
| Y           | Normal: Operation has succeeded / When operation code was U: Record updated |
| X           | Normal: Operation has failed / When operation code was U: Record added      |
| U           | Unknown operation code has been supplied                                    |

## DPCExample class

In this section, we investigate the key methods of the DPCExample class.

### Instance variables

The following instance variables are declared for the class:

```
private AS400 as400;
private ProgramCall pgm;
private String progName = "/QSYS.LIB/apilib.LIB/DPCXRPG.PGM";
private ToolboxGUI ivjToolboxGUI1 = null;
```

### **The connectToDB method**

The connectToDB method is called by the ToolboxGUI class when the Connect button is clicked. String parameters representing the iSeries server name, user ID, and password are passed to the method.

---

#### *Example 4-24 Distributed program call example connectToDB method*

---

```
public void connectToDB(String systemName, String userid, String password) throws Exception
{
    as400 = new AS400(systemName, userid, password);
    as400.connectService(AS400.COMMAND);
    pgm = new ProgramCall(as400); // pgm is declared elsewhere
    return;
}
```

---

**Class:** DPCEExample

**Method:** connectToDB

The connectToDB method implements these functions:

```
as400 = new AS400(systemName, userid, password);
```

- ▶ Creates a new AS/400 connection object. System name, user ID, and password are passed through the constructor.

```
as400.connectService(AS400.COMMAND);
```

- ▶ Connects to the iSeries program call and command call server. This is not required. If a service connection is needed and does not already exist, the service is connected automatically. We choose to place the connection overhead in the connect method as opposed to connecting the first time the user requests a part record.

```
pgm = new ProgramCall(as400);
```

- ▶ Creates a new ProgramCall object for the iSeries server defined in the AS400 object. The program and parameter information are supplied later. The pgm object is declared elsewhere in this class.

### **The getRecord method**

The getRecord method is called by the ToolboxGUI class when the Get Part button is clicked.

---

#### *Example 4-25 Distributed program call example getRecord method*

---

```
public String getRecord(String partNo, javax.swing.JTextField partDesc,
javax.swing.JTextField partQty, javax.swing.JTextField partPrice, javax.swing.JTextField
partDate) throws Exception
{
    ProgramParameter[] parmlist = new ProgramParameter[6];
    AS400Text asFlag = new AS400Text(1,as400);
    parmlist[0] = new ProgramParameter(asFlag.toBytes("S"), 1);
    AS400PackedDecimal asPartNo = new AS400PackedDecimal(5, 0);
    parmlist[1] = new ProgramParameter(asPartNo.toBytes(new
        java.math.BigDecimal(partNo.trim()))),3);
    AS400Text asDesc = new AS400Text(25,as400);
    parmlist[2] = new ProgramParameter(asDesc.toBytes(""), 25);
    AS400PackedDecimal asQty = new AS400PackedDecimal(5, 0);
    parmlist[3] = new ProgramParameter(asQty.toBytes(new java.math.BigDecimal(0)), 3);
```

```

AS400PackedDecimal asPrice = new AS400PackedDecimal(6, 2);
parmlist[4] = new ProgramParameter(asPrice.toBytes(new java.math.BigDecimal(0)), 4);
AS400Text asDate = new AS400Text(10,as400);
parmlist[5] = new ProgramParameter(asDate.toBytes("0001-01-01"), 10);
pgm.setProgram(progName, parmlist);
if (pgm.run() != true)
{
    System.out.println("program failed:" + progName);
    AS400Message[] messagelist = pgm.getMessageList();
    for (int i = 0; i < messagelist.length; i++)
    {
        System.out.println(messagelist[i]);
    }
    return "Program call failed!";
}
else
{
    if (((String) (asFlag.toObject(parmlist[0].getOutputData(), 0))).equals("Y"))
    {
        partDesc.setText(((String) (new
            AS400Text(25,as400)).toObject(parmlist[2].getOutputData(), 0)).trim());
        partQty.setText(((java.math.BigDecimal) (new AS400PackedDecimal(5,
            0)).toObject(parmlist[3].getOutputData(), 0)).toString());
        partPrice.setText(((java.math.BigDecimal) (new AS400PackedDecimal(6,
            2)).toObject(parmlist[4].getOutputData(), 0)).toString());
        partDate.setText((String) (new
            AS400Text(10,as400)).toObject(parmlist[5].getOutputData(), 0));
        return "Record found.";
    }
    else
    {
        partDesc.setText("");
        partQty.setText("0");
        partPrice.setText("0.00");
        partDate.setText("");
        return "Record not found.";
    }
}
}

```

---

**Class:** DPCExample

**Method:** getRecord

Let's look more closely at the getRecord method:

```
ProgramParameter[] parmlist = new ProgramParameter[6];
```

- Declares a ProgramParameter array for six parameters.

```
AS400Text asFlag = new AS400Text(1,as400);
parmlist[0] = new ProgramParameter(asFlag.toBytes("S"), 1);
```

- The first parameter is an action code of one character. An object of type AS400Text with a length of one is created and called asFlag. The asFlag object is used to convert a Java String object with a value of S to its iSeries equivalent and returned as an array of bytes. This byte array is used as the input for a program parameter. The second argument of the ProgramParameter constructor is an integer declaring the number of bytes expected to be returned by the program after execution.

```
AS400PackedDecimal asPartNo = new AS400PackedDecimal(5, 0);
parmlist[1] = new ProgramParameter(asPartNo.toBytes(new
java.math.BigDecimal(partNo.trim())), 3);
```

- ▶ The second parameter is a part number and is both input and output. An AS400PackedDecimal conversion object is created to convert the part number to its iSeries format. An output buffer of three bytes is reserved for the value returned by the called program.

```
AS400Text asDesc = new AS400Text(25,as400);
parmlist[2] = new ProgramParameter(asDesc.toBytes("", 25);
AS400PackedDecimal asQty = new AS400PackedDecimal(5, 0);
parmlist[3] = new ProgramParameter(asQty.toBytes(new java.math.BigDecimal(0)),3);
AS400PackedDecimal asPrice = new AS400PackedDecimal(6, 2);
parmlist[4] = new ProgramParameter(asPrice.toBytes(new
java.math.BigDecimal(0)), 4);
AS400Text asDate = new AS400Text(10,as400);
parmlist[5] = new ProgramParameter(asDate.toBytes("0001-01-01"), 10);
```

- ▶ The next four parameters are used for output only, but require the correct formats to be set. This avoids errors in the server program, such as decimal data errors.

```
pgm.setProgram(progName, parmlist);
```

- ▶ Associates a program name and parameter list with the ProgramCall object.

```
if (pgm.run() != true)
```

- ▶ Uses the run method of the ProgramCall object to execute the program on the iSeries server. The method returns a true value if successful and a false value if a problem occurred.

```
AS400Message[] messagelist = pgm.getMessageList();
for (int i = 0; i < messagelist.length; i++)
{System.out.println(messagelist[i]);}
```

- ▶ If an error occurred on the run() method, obtain the error messages from the ProgramCall object and print each message on the console.

```
if (((String) (asFlag.toObject(parmlist[0].getOutputData(), 0))).equals("Y"))
```

- ▶ This statement checks the value of the action parameter returned by the program to see if the part record was retrieved successfully. parmlist[0].getOutputData() returns an array of bytes for the first parameter in the iSeries format. The toObject method is used on the AS400Text object, asFlag, to convert the byte array to a Java object. Since toObject returns an object of type Object, it must be typecast as a string object to use string methods.

```
partDesc.setText((String) (new
    AS400Text(25,as400)).toObject(parmlist[2].getOutputData(),0)).trim());
partQty.setText(((java.math.BigDecimal) (new
    AS400PackedDecimal(5,0)).toObject(parmlist[3].getOutputData(), 0)).toString());
partPrice.setText(((java.math.BigDecimal) (new
    AS400PackedDecimal(6,2)).toObject(parmlist[4].getOutputData(), 0)).toString());
partDate.setText((String) (new AS400Text(10,as400)).toObject(parmlist[5].getOutputData(),
0));
```

- The same technique is used to retrieve and convert parameter values from the iSeries format to Java objects. The string representation of each output parameter is used to set the text property of the associated JTextFields on the window.

### **The populateAllParts method**

The populateAllParts method is called from the DisplayAllParts through the PartsContainer Interface. It runs the RPG program multiple times to retrieve all records from the PARTS file.

*Example 4-26 Distributed program call example populateAllParts method*

---

```

public void populateAllParts(javax.swing.table.DefaultTableModel defaultTableModel) throws
Exception {
    ProgramParameter[] parmlist = new ProgramParameter[6];
    AS400Text asFlag = new AS400Text(1,as400);
    parmlist[0] = new ProgramParameter(asFlag.toBytes("A"), 1);
    AS400PackedDecimal asPartNo = new AS400PackedDecimal(5, 0);
    parmlist[1] = new ProgramParameter(asPartNo.toBytes(new java.math.BigDecimal(0)), 3);
    AS400Text asDesc = new AS400Text(25,as400);
    parmlist[2] = new ProgramParameter(asDesc.toBytes("", 25));
    AS400PackedDecimal asQty = new AS400PackedDecimal(5, 0);
    parmlist[3] = new ProgramParameter(asQty.toBytes(new java.math.BigDecimal(0)), 3);
    AS400PackedDecimal asPrice = new AS400PackedDecimal(6, 2);
    parmlist[4] = new ProgramParameter(asPrice.toBytes(new java.math.BigDecimal(0)), 4);
    AS400Text asDate = new AS400Text(10,as400);
    parmlist[5] = new ProgramParameter(asDate.toBytes("0001-01-01"), 10);
    pgm.setProgram(progName, parmlist);
    String flag = null;
    if (pgm.run() != true) {
        System.out.println("program failed:" + progName);
        AS400Message[] messagelist = pgm.getMessageList();
        for (int i = 0; i < messagelist.length; i++) {
            System.out.println(messagelist[i]);
        }
        return;
    } else {
        flag = (String) (asFlag.toObject(parmlist[0].getOutputData(), 0));
        if (flag.equals("Y")) {
            parmlist[0] = new ProgramParameter(asFlag.toBytes("F"), 1);
            pgm.setProgram(progName, parmlist);
            do {
                if (pgm.run() != true) {
                    System.out.println("program failed:" + progName);
                    AS400Message[] messagelist = pgm.getMessageList();
                    for (int i = 0; i < messagelist.length; i++) {
                        System.out.println(messagelist[i]);
                    }
                }
                return;
            } else {
                flag = (String) (asFlag.toObject(parmlist[0].getOutputData(), 0));
                if (flag.equals("Y")) {
                    String[] array = new String[5];
                    array[0] = (((java.math.BigDecimal) (new AS400PackedDecimal(5, 0)).toObject
                    (parmlist[1].getOutputData(), 0))).toString();
                    array[1] = (String) (new
                    AS400Text(25,as400)).toObject(parmlist[2].getOutputData(),0);
                    array[2] = DisplayAllParts.insertSpaces(((java.math.BigDecimal) (new
                    AS400PackedDecimal(5, 0)).toObject(parmlist[3].getOutputData(),
                    0)).toString(),5);
                    array[3] = DisplayAllParts.insertSpaces(((java.math.BigDecimal) (new

```

```

        AS400PackedDecimal(6, 2)).toObject(parmlist[4].getOutputData(),
        0)).toString(),8);
array[4] = (String) (new
AS400Text(10,as400)).toObject(parmlist[5].getOutputData(),0);
defaultTableModel.addRow(array);
    }
} while (flag.equals("Y"));
}
return;
}

```

---

**Class:** DPCExample

**Method:** populateAllParts

The method is highlighted here:

```

AS400Text asFlag = new AS400Text(1,as400);
parmList[0] = new ProgramParameter(asFlag.toBytes("A"), 1);
AS400PackedDecimal asPartNo = new AS400PackedDecimal(5, 0);
parmList[1] = new ProgramParameter(asPartNo.toBytes(new
java.math.BigDecimal(0)), 3);

```

- ▶ The program parameter list is defined and initialized in the same manner as in the getRecord method. Here, the first parameter is set to **A** to tell the program to retrieve all records from the parts file. Zero is supplied for the part number.

```
if (pgm.run() != true)
```

- ▶ Calls the program the first time to open the parts file and position the read pointer to the first record in the file.

```
flag = (String)(asFlag.toObject(parmList[0].getOutputData(),0));
if (flag.equals(Y))
```

- ▶ Checks if there was any record at all to retrieve. If the initial call to the program was successful, retrieve the value of the first parameter and check for an *operation succeeded* code (Y).

```
parmList[0] = new ProgramParameter( asFlag.toBytes(" F") , 1);
```

- ▶ Change the value of the first parameter to an **F** to tell the program to retrieve the next record from the file.

Execute the program inside a do loop until the value returned in the first parameter is not a **Y**. This means that there are no more records to retrieve from the file. Upon each successful call to the program, use the same techniques as in the getRecord method to retrieve the values of output parameters. Place the Java String converted value into a string array for an addition to the default table model.

### ***The updateRecord method***

The updateRecord method is called by the ToolboxGUI class when the Update/Add Part button is clicked.

---

*Example 4-27 Distributed program call example updateRecord method*

---

```
public String updateRecord(String partNo, String partDesc, String partQty, String
partPrice, String partDate) throws Exception
{
    ProgramParameter[] parmlist = new ProgramParameter[6];
    AS400Text asFlag = new AS400Text(1,as400);
    parmlist[0] = new ProgramParameter(asFlag.toBytes("U"), 1);
    AS400PackedDecimal asPartNo = new AS400PackedDecimal(5, 0);
    parmlist[1] = new ProgramParameter(asPartNo.toBytes(new
        java.math.BigDecimal(partNo.trim())),3);
    AS400Text asDesc = new AS400Text(25,as400);
    parmlist[2] = new ProgramParameter(asDesc.toBytes(partDesc), 25);
    AS400PackedDecimal asQty = new AS400PackedDecimal(5, 0);
    parmlist[3] = new ProgramParameter(asQty.toBytes(new java.math.BigDecimal(partQty)), 3);
    AS400PackedDecimal asPrice = new AS400PackedDecimal(6, 2);
    parmlist[4] = new ProgramParameter(asPrice.toBytes(new java.math.BigDecimal(partPrice)), 4);
    AS400Text asDate = new AS400Text(10,as400);
    parmlist[5] = new ProgramParameter(asDate.toBytes(partDate), 10);
    pgm.setProgram(progName, parmlist);
    if (pgm.run() != true)
    {
        System.out.println("program failed:" + progName);
        AS400Message[] messagelist = pgm.getMessageList();
        for (int i = 0; i < messagelist.length; i++)
        {System.out.println(messagelist[i]);}
        return "Program call failed!";
    }
    else
    {
        if (((String) (asFlag.toObject(parmlist[0].getOutputData(), 0))).equals("Y"))
        {return "Record updated.";}
        else
        {return "Record added.";}
    }
}
```

---

**Class:** DPCExample

**Method:** updateRecord

The updateRecord method is highlighted here:

```
AS400Text asFlag = new AS400Text(1,as400);
parmlist[0] = new ProgramParameter(asFlag.toBytes("U"), 1);
AS400PackedDecimal asPartNo = new AS400PackedDecimal(5, 0);
parmlist[1] = new ProgramParameter(asPartNo.toBytes(new
    java.math.BigDecimal(partNo.trim())), 3);
AS400Text asDesc = new AS400Text(25,as400);
parmlist[2] = new ProgramParameter(asDesc.toBytes(partDesc), 25);
AS400PackedDecimal asQty = new AS400PackedDecimal(5, 0);
parmlist[3] = new ProgramParameter(asQty.toBytes(new
    java.math.BigDecimal(partQty)), 3);
AS400PackedDecimal asPrice = new AS400PackedDecimal(6, 2);
parmlist[4] = new ProgramParameter(asPrice.toBytes(new
    java.math.BigDecimal(partPrice)), 4);
AS400Text asDate = new AS400Text(10,as400);
parmlist[5] = new ProgramParameter(asDate.toBytes(partDate), 10);
```

Here, the first parameter is set to **U** to tell the program to retrieve a single record by part number, update it with the supplied attribute values, and return a **Y**, meaning that the record was updated. If the record cannot be found in the database file, the program writes it into the database file. The record is written with all the supplied data and the program returns an **X**, meaning that the record was added.

#### **The deleteRecord method**

The deleteRecord method is called by the ToolboxGUI class when the Delete Part button is clicked. It works in the same way as the getRecord method, but a **D** is supplied as the operation code.

#### **The dispose method**

The dispose method is called when the application window is closed.

*Example 4-28 Distributed program call example dispose method*

---

```
public void dispose()
{
    try
    {
        as400.disconnectAllServices();
    }
    catch (Exception e)
    {
    };
    super.dispose();
    System.exit(0);
    return;
}
```

---

**Class:** DPCEexample

**Method:** dispose

The dispose method supports:

```
as400.disconnectAllServices();
```

Releases all connections to the iSeries server and releases resources associated with server jobs processing requests for the client.

All other methods used in the dispose method are the same as in the JDBC and StoredProcedure examples.

### **4.6.11 Data queues**

Data queues are iSeries objects which can be used to store information. They are commonly used to pass data from one application to another.

The Toolbox DataQueue classes allow a Java program to create, delete, write, and read data queues on an iSeries server. They allow a Java program to interact with iSeries data queues. iSeries data queues have the following characteristics:

- ▶ The data queue is a fast means of communications between jobs. Therefore, it is an excellent way to synchronize and pass data between jobs.
- ▶ Many jobs can access them simultaneously.
- ▶ Messages on a data queue are in free format. Fields are not required as in database files.

- ▶ The data queue can be used for either synchronous or asynchronous processing.
- ▶ The messages on a data queue can be ordered in one of three ways:
  - **Last in, first out (LIFO)**: The last (newest) message placed on the data queue is the first message taken off the queue.
  - **First in, first out (FIFO)**: The first (oldest) message placed on the data queue is the first message taken off the queue.
  - **Keyed**: Each message on the data queue has a key associated with it. A message can only be taken off the queue by specifying the key that is associated with it.
- ▶ Data queues allow for time independent applications. The client and server applications are not communicating directly and can work independent of each other.

The DataQueue class provides a complete set of interfaces to access iSeries data queues from a Java program. It is an excellent way to communicate between Java programs and iSeries programs. The iSeries program can be written in any language.

A required parameter of the DataQueue constructor is the AS400 object that represents the iSeries server that has the data queue or where the data queue is to be created. The DataQueue constructor requires the integrated file system path name of the data queue.

Two types of data queues are supported: keyed and non-keyed. Methods common to both types of queues are in the BaseDataQueue class. This class is extended by the DataQueue class to complete the implementation of non-keyed data queues. The BaseDataQueue class is extended by the KeyedDataQueue class to complete the implementation of keyed data queues.

When data is read from a data queue, it is placed in a DataQueueEntry object. This object holds the data for both keyed and non-keyed data queues. Additional data available when reading from a keyed data queue is placed in a KeyedDataQueueEntry object that extends the DataQueueEntry class. Consider this example:

```
// Create an AS400 object
AS400 sys = new AS400("mySystem.myCompany.com");

// Create the DataQueue object
DataQueue dq = new DataQueue(sys, "/QSYS.LIB/MYLIB.LIB/MYQUEUE.DTAQ");

// read data from the queue
DataQueueEntry dqData = dq.read();

// get the data out of the DataQueueEntry object as a byte array
byte[] data = dqData.getByteData();

// ... process the data

// Disconnect since I am done using data queues
sys.disconnectService("data queue");
```

The data queue classes do not alter data written to or read from the iSeries data queue. It is up to the Java program to correctly format the data. The data conversion classes provide methods for converting data.

## Keyed data queues

The BaseDataQueue and KeyedDataQueue classes provide the following methods for working with keyed data queues:

- ▶ Create a keyed data queue on the iSeries server. The Java program must specify a key length and maximum size of an entry on the queue. The Java program can optionally specify authority information, save sender information, force to disk, and provide a queue description.
- ▶ Peek at an entry that matches the specified key without removing it from the queue. The Java program can wait or return immediately if no entry is currently on the queue that matches the specified key. The program can receive the entry as a string or as a byte array.
- ▶ Read an entry off the queue that matches the specified key. The Java program can wait or return immediately if no entry is available on the queue that matches the specified key. The program can read the entry as a string or as a byte array.
- ▶ Write an entry to the queue.
- ▶ Clear all entries that match the specified key.
- ▶ Delete the queue.

The BaseDataQueue and KeyedDataQueue classes also provide additional methods for retrieving the attributes of the data queue.

### **Non-keyed data queues**

Entries on a non-keyed iSeries data queue are removed in FIFO or LIFO sequence. The BaseDataQueue and DataQueue classes provide the following methods for working with non-keyed data queues:

- ▶ Create a data queue on the iSeries server. The Java program can optionally specify queue parameters (FIFO versus LIFO, save sender information, and so on) when the queue is created.
- ▶ Peek at an entry on the data queue without removing it from the queue. The Java program can wait or return immediately if no entry is currently on the queue. It can also receive the entry as a string or as a byte array.
- ▶ Read an entry off the queue. The Java program can wait or return immediately if no entry is available on the queue. It can also read the entry as a string or as a byte array.
- ▶ Write an entry to the queue.
- ▶ Clear all entries from the queue.
- ▶ Delete the queue.

The BaseDataQueue and DataQueue classes also provide additional methods for retrieving the attributes of the data queue.

### **4.6.12 Data queue application example**

In this example, we use the data queue interface to allow a client program to interface with an iSeries program (see Figure 4-16).

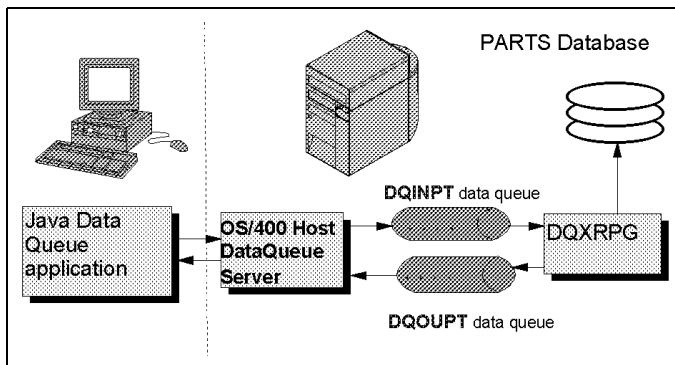


Figure 4-16 Data queue application

The client program requests data from the iSeries database by placing requests on an input iSeries data queue. A host program monitors the input data queue for a request. If a request is received, the host program uses it to retrieve records from the iSeries database. The output information is placed in an output data queue that is monitored by the client program. When using data queues, it is up to the application implementer to handle data conversions.

Class DataQueueExample is the main class in this application. It is functionally equivalent to the StoredProcedureExample application, but is implemented using different techniques. The ToolboxGUI and DisplayAllParts classes are used to handle all user interaction. A native RPG program waits for a request on an input data queue (APILIB/DQINPT) and places results on an output data queue (APILIB/DQOUTP). Record create, read, update, and delete are implemented in this example.

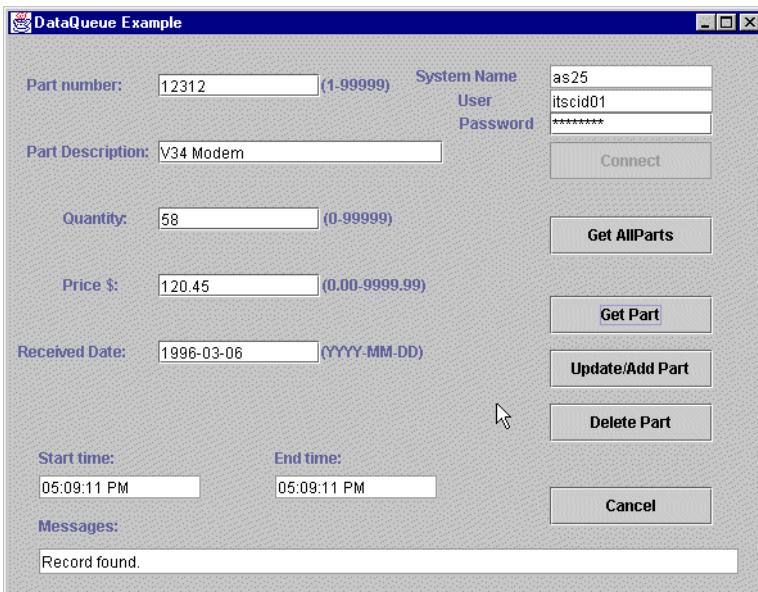


Figure 4-17 DataQueue Example display

### Data queue server program background

An input queue and output queue were created with these commands:

```
CRTDTAQ DTAQ(APILIB/DQINPT) MAXLEN(48) TEXT('Data Queue for Parts Input')
CRTDTAQ DTAQ(APILIB/DQOUTP) MAXLEN(48) TEXT('Data Queue for Parts Output')
```

Table 4-23 shows the DQINPT layout for the data queue.

*Table 4-23 Data queue DQINPT layout*

| Queue position | Description        | Length/type  | Values                                                                                                                            |
|----------------|--------------------|--------------|-----------------------------------------------------------------------------------------------------------------------------------|
| 1 - 1          | Operation Code     | 1 character  | S - Read single record<br>A - Read all records<br>E - End the program<br>D - Delete single record<br>U - Update/add single record |
| 2 - 6          | Part Number        | 5.0 zoned    |                                                                                                                                   |
| 7 - 31         | Part Description   | 25 character |                                                                                                                                   |
| 32 - 34        | Part Quantity      | 5.0 packed   |                                                                                                                                   |
| 35 - 38        | Part Price         | 6.2 packed   |                                                                                                                                   |
| 39 - 48        | Part Date Received | 10 date      |                                                                                                                                   |

Table 4-24 shows the DQOUPT layout for the data queue.

*Table 4-24 Data queue DQOUPT layout*

| Queue position | Description        | Length/type  | Values                                                                         |
|----------------|--------------------|--------------|--------------------------------------------------------------------------------|
| 1 - 1          | Return Code        | 1 character  | Y - record found, deleted, updated<br>X - record not found, added, End of File |
| 2 - 6          | Part Number        | 5.0 zoned    |                                                                                |
| 7 - 31         | Part Description   | 25 character |                                                                                |
| 32 - 34        | Part Quantity      | 5.0 packed   |                                                                                |
| 35 - 38        | Part Price         | 6.2 packed   |                                                                                |
| 39 - 48        | Part Date Received | 10 date      |                                                                                |

## **DataQueueExample class**

This section investigates the key methods of the DataQueueExample class.

### ***Instance variables***

The following instance variables are declared for the class:

```
private AS400 as400;  
private DataQueue dqInput;  
private DataQueue dqOutput;  
private RecordFormat rfInput;  
private RecordFormat rfOutput;
```

### ***The connectToDB method***

The connectToDB method is called by the ToolboxGUI class when the Connect button is clicked. String parameters representing the iSeries server name, user ID, and password are passed to the method.

---

*Example 4-29 Data queue example connectToDB method*

---

```
public void connectToDB(String systemName, String userid, String password) throws Exception
{
    as400 = new AS400(systemName, userid, password);
    dqInput = new com.ibm.as400.access.DataQueue(as400, "/QSYS.LIB/APILIB.LIB/DQINPT.DTAQ");
    dqOutput = new com.ibm.as400.access.DataQueue(as400,
        "/QSYS.LIB/APILIB.LIB/DQOUTP.DTAQ");
    dqInput.clear();
    dqOutput.clear();
    return;
}
```

---

**Class:** DataQueueExample

**Method:** connectToDB

Let's take a closer look at this method:

```
as400 = new com.ibm.as400.access.AS400(systemName, userid, password);
```

- ▶ Creates a new AS/400 connection object. System name, user ID, and password are passed through the constructor.

**Note:** If you import the com.ibm.as400.access classes, you do not have to fully qualify the class name. You can write it as shown.

```
dqInput = new com.ibm.as400.access.DataQueue(as400,
"/QSYS.LIB/APILIB.LIB/DQINPT.DTAQ");
dqOutput = new com.ibm.as400.access.DataQueue(as400,
"/QSYS.LIB/APILIB.LIB/DQOUTP.DTAQ");
```

- ▶ Creates new DataQueue objects for the input and output queues. The fully-qualified IFS name of the data queues are passed in the constructor.

```
dqInput.clear();
dqOutput.clear();
```

- ▶ Clears both DataQueues, so there are no remaining entries from interrupted or unsuccessful tests. DataQueues that are not properly initialized can provoke confusing and unwanted results in your application.

### ***The getRecord method***

The getRecord method is called by the ToolboxGUI class when the Get Part button is clicked.

---

*Example 4-30 Data queue example getRecord method*

---

```
public String getRecord(String partNo, javax.swing.JTextField partDesc,
javax.swing.JTextField partQty, javax.swing.JTextField partPrice, javax.swing.JTextField
partDate) throws Exception
{
    if (rfInput == null) initRecordFormat();
    Record rInput = rfInput.getNewRecord();
    rInput.setField("flag","S");
    rInput.setField("partno",new java.math.BigDecimal(partNo.trim()));
    rInput.setField("partds","");
}
```

```

rInput.setField("partqy",new java.math.BigDecimal(0));
rInput.setField("partpr",new java.math.BigDecimal(0));
rInput.setField("partdt","0001-01-01");
dqInput.write(rInput.getContents());
DataQueueEntry dqe = null;
while (dqe == null)
{
    dqe = dqOutput.read();
}
Record rOutput = rfOutput.getNewRecord(dqe.getData());
if (((String)rOutput.getField("flag")).equals("Y"))
{
    partDesc.setText(((String)rOutput.getField("partds")).trim());
    partQty.setText(((java.math.BigDecimal)rOutput.getField("partqy")).toString());
    partPrice.setText(((java.math.BigDecimal)rOutput.getField("partpr")).toString());
    partDate.setText((String)rOutput.getField("partdt"));
}
else
{
    partDesc.setText("");
    partQty.setText("0");
    partPrice.setText("0.00");
    partDate.setText("");
    return "Record not found.";
}
return "Record found.";
}

```

---

**Class:** DataQueueExample

**Method:** getRecord

The highlights of this method are listed here:

```
if (rfInput == null) initRecordFormat();
```

- ▶ Uses lazy initialization to create the input and output record format objects. See the `initRecordFormat` method for details.

```
Record rInput = rfInput.getNewRecord();
```

- ▶ Creates a new input record object from the input record format. A RecordFormat is only a description of a record. A record is an object that can have field values.

```

rInput.setField("flag","S");
rInput.setField("partno",new java.math.BigDecimal(partNo.trim()));
rInput.setField("partds","");
rInput.setField("partqy",new java.math.BigDecimal(0));
rInput.setField("partpr",new java.math.BigDecimal(0));
rInput.setField("partdt","0001-01-01");

```

- ▶ Sets the value of the flag field in the record to S to tell the server program to retrieve a single record from the database. Set the value of the part field to the part number passed on the parameter list. Initialize the remaining fields with values accepted by the server program. This has to be done because the input parameters must match the requirements of the iSeries data types and avoids having such server program troubles as decimal data errors or date conversion errors.

```
dqInput.write(rInput.getContents());
▶ Writes the input record to the input data queue. The getContents method returns a byte array of the value of the record in the iSeries format.
```

```
DataQueueEntry dqe = null;
while (dqe == null) {dqe = dqOutput.read();}

▶ Initializes a DataQueueEntryObject. In a loop, the dqOutput.read() method reads the next entry off the output data queue. This returns a data queue entry object. Since we do not know how long it will take the server program to write the answer of our request into the output dataQueue, the program must loop until the expected record can be retrieved from there.
```

```
Record rOutput = rfOutput.getNewRecord(dqe.getData());
▶ Creates a new output record by using the output record format and setting field values that use the array of bytes returned by the getData method of the data queue entry object.
```

```
if (((String)rOutput.getField("flag")).equals("Y"))
▶ This statement checks the value of the flag field in the record format to see if the part record was retrieved successfully. The getField method uses an object of type Object. It must be typecast as a string object to use string methods.
```

```
partDesc.setText(((String)rOutput.getField("partds")).trim());
partQty.setText(((java.math.BigDecimal)rOutput.getField("partqy")).toString());
partPrice.setText(((java.math.BigDecimal)rOutput.getField("partpr")).toString());
partDate.setText((String)rOutput.getField("partdt"));
```

```
▶ The same technique is used to retrieve and field values from the output record object to Java objects. The string representation of each output parameter is used to set the text property of the associated JTextFields on the window.
```

### ***The initRecordFormat method***

The initRecordFormat method initializes the input and output record format objects. It is called by the getRecord and populateAllParts methods if the record formats are not already initialized.

---

#### *Example 4-31 Data queue example initRecordFormat method*

---

```
public void initRecordFormat()
{
    CharacterFieldDescription asFlag = new CharacterFieldDescription(new
AS400Text(1,as400),"flag");
    ZonedDecimalFieldDescription asPartNo = new ZonedDecimalFieldDescription(new
AS400ZonedDecimal(5,0),"partno");
    CharacterFieldDescription asPartDS = new CharacterFieldDescription(new
AS400Text(25,as400),"partds");
    PackedDecimalFieldDescription asPartQy = new PackedDecimalFieldDescription(new
AS400PackedDecimal(5,0),"partqy");
    PackedDecimalFieldDescription asPartPR = new PackedDecimalFieldDescription(new
AS400PackedDecimal(6,2),"partpr");
    DateFieldDescription asPartDt = new DateFieldDescription(new
AS400Text(10,as400),"partdt");
    rfInput = new RecordFormat();
    rfInput.addFieldDescription(asFlag);
    rfInput.addFieldDescription(asPartNo);
```

```

rfInput.addFieldDescription(asPartDS);
rfInput.addFieldDescription(asPartQy);
rfInput.addFieldDescription(asPartPR);
rfInput.addFieldDescription(asPartDt);
rfOutput = new RecordFormat();
rfOutput.addFieldDescription(asFlag);
rfOutput.addFieldDescription(asPartNo);
rfOutput.addFieldDescription(asPartDS);
rfOutput.addFieldDescription(asPartQy);
rfOutput.addFieldDescription(asPartPR);
rfOutput.addFieldDescription(asPartDt);
return;
}

```

---

**Class:** DataQueueExample

**Method:** initRecordFormat

The highlights of this method are listed here:

```

CharacterFieldDescription asFlag = new CharacterFieldDescription(new
AS400Text(1,as400),"flag");
ZonedDecimalFieldDescription asPartNo = new ZonedDecimalFieldDescription(new
AS400ZonedDecimal(5,0),"partno");
CharacterFieldDescription asPartDS = new CharacterFieldDescription(new
AS400Text(25,as400),"partds");
PackedDecimalFieldDescription asPartQy = new PackedDecimalFieldDescription(new
AS400PackedDecimal(5,0),"partqy");
PackedDecimalFieldDescription asPartPR = new PackedDecimalFieldDescription(new
AS400PackedDecimal(6,2),"partpr");
DateFieldDescription asPartDt = new DateFieldDescription(new
AS400Text(10,as400),"partdt");

```

- ▶ These statements create field description objects for the data fields that make up the input and output record formats. The field description constructor takes an iSeries data type object and a field name.

```

rfInput = new RecordFormat();
rfInput.addFieldDescription(asFlag);
rfInput.addFieldDescription(asPartNo);
rfInput.addFieldDescription(asPartDS);
rfInput.addFieldDescription(asPartQy);
rfInput.addFieldDescription(asPartPR);
rfInput.addFieldDescription(asPartDt);

```

- ▶ Creates the input record format by adding field descriptions to a new RecordFormat object.

```

rfOutput = new RecordFormat();
rfOutput.addFieldDescription(asFlag);
rfOutput.addFieldDescription(asPartNo);
rfOutput.addFieldDescription(asPartDS);
rfOutput.addFieldDescription(asPartQy);
rfOutput.addFieldDescription(asPartPR);
rfOutput.addFieldDescription(asPartDt);

```

- ▶ Creates the output record format by adding field descriptions to a new RecordFormat object.

### **The populateAllParts method**

The populateAllParts method is called from the Display All Parts button. It sends a message on the input data queue to request all records from the parts file. It receives from the output data queue multiple times until all of the parts records are returned by the server program.

---

#### *Example 4-32 Data queue example populateAllParts method*

---

```
public void populateAllParts(javax.swing.table.DefaultTableModel defaultTableModel) throws
Exception
{
    if (rfInput == null) initRecordFormat();
    Record rInput = rfInput.getNewRecord();
    rInput.setField("flag","A");
    rInput.setField("partno",new java.math.BigDecimal(0));
    rInput.setField("partds","");
    rInput.setField("partqy",new java.math.BigDecimal(0));
    rInput.setField("partpr",new java.math.BigDecimal(0));
    rInput.setField("partdt","0001-01-01");
    dqInput.write(rInput.getContents());
    String flag = null;
    do
    {
        DataQueueEntry dqe = null;
        while (dqe == null)
        {
            dqe = dqOutput.read();
        }
        Record rOutput = rfOutput.getNewRecord(dqe.getData());
        flag = (String)rOutput.getField("flag");
        if (flag.equals("Y"))
        {
            String[] array = new String[5];
            array[0] =((java.math.BigDecimal)rOutput.getField("partno")).toString();
            array[1] =(String)rOutput.getField("partds");
            array[2] =DisplayAllParts.insertSpaces(((java.math.BigDecimal)
                rOutput.getField("partqy")).toString(), 5);
            array[3] =DisplayAllParts.insertSpaces(((java.math.BigDecimal)
                rOutput.getField("partpr")).toString(), 8);
            array[4] =(String)rOutput.getField("partdt");
            defaultTableModel.addRow(array);
        }
    }
    while (flag.equals("Y"));
    return;
}
```

---

**Class:** DataQueueExample

**Method:** populateAllParts

We examine this method here:

```
if (rfInput == null) initRecordFormat();
► The input and output record formats are initialized, if needed.
```

```
Record rInput = rfInput.getNewRecord();
```

► A new input record object is created from the input record format.

```
rInput.setField("flag", "A");
▶ The flag field in the input record is set to an "A" to request all records from the parts file.
```

```
dqInput.write(rInput.getContents());
▶ Puts the current value of the input record format on the input data queue.
```

```
DataQueueEntry dqe = null;
while (dqe == null) {dqe = dqOutput.read();}
```

```
▶ Initializes DataQueueEntryObject. In a loop, the dqOutput.read() method reads the next entry off the output data queue. This returns a data queue entry object. Since we do not know how long it will take the server program to write the answer of our request into the output dataQueue, the program must loop until the expected record can be retrieved from there.
```

```
Record rOutput = rfOutput.getNewRecord(dqe.getData());
```

```
▶ Uses the array of bytes returned by the getData() method to initialize a new output record object.
```

Execute the read inside a do loop until the value returned in the flag field is not a "Y". This means that there are no more records to retrieve from the file. Upon each successful read from the data queue, use the same techniques as the getRecord method to retrieve the values of output record fields. Place the Java String converted value into a string array for addition to the default table model.

### ***The updateRecord method***

The updateRecord method is called by the ToolboxGUI class when the Update/Add Part button is clicked.

---

#### *Example 4-33 Data queue example updateRecord method*

---

```
public String updateRecord(String partNo, String partDesc, String partQty, String
partPrice, String partDate) throws Exception
{
    if (rfInput == null) initRecordFormat();
    Record rInput = rfInput.getNewRecord();
    rInput.setField("flag", "U");
    rInput.setField("partno", new java.math.BigDecimal(partNo.trim()));
    rInput.setField("partds", partDesc);
    rInput.setField("partqy", new java.math.BigDecimal(partQty));
    rInput.setField("partpr", new java.math.BigDecimal(partPrice));
    rInput.setField("partdt", partDate);
    dqInput.write(rInput.getContents());
    DataQueueEntry dqe = null;
    while (dqe == null)
    {
        dqe = dqOutput.read();
    }
    Record rOutput = rfOutput.getNewRecord(dqe.getData());
```

```

        if (((String) rOutput.getField("flag")).equals("Y"))
        {
            return "1 Record updated.";
        }
        else
        {
            return "1 Record added.";
        }
    }

```

---

**Class:** DataQueueExample

**Method:** updateRecord

The updateRecord method is presented here:

```

rInput.setField("flag", "U");
rInput.setField("partno", new java.math.BigDecimal(partNo.trim()));
rInput.setField("partds", partDesc);
rInput.setField("partqy", new java.math.BigDecimal(partQty));
rInput.setField("partpr", new java.math.BigDecimal(partPrice));
rInput.setField("partdt", partDate);

```

Here, the first parameter is set to **U** to tell the program to retrieve a single record by its part number, update it with the supplied attribute values, and return a **Y**, meaning that the record was updated. If the record cannot be found in the database file, the program writes it into the database file. The record is written with all the supplied data and the program returns an **X**, meaning that the record has been added.

#### ***The deleteRecord method***

The deleteRecord method is called by the ToolboxGUI class when the Delete Part button is clicked. It works in the same way as the getRecord method, but a **D** is supplied as the operation code.

#### ***The dispose method***

The dispose method is called when the application window is closed.

---

*Example 4-34 Data queue example dispose method*

---

```

public void dispose()
{
    try
    {
        as400.disconnectAllServices();
    }
    catch (Exception e)
    {
    };
    super.dispose();
    System.exit(0);
    return;
}

```

---

**Class:** DataQueueExample

**Method:** dispose

The dispose method includes:

```
as400.disconnectAllServices();
```

- ▶ Releases all connections to the iSeries server and releases resources associated with server jobs processing requests for the client.

```
super.dispose();
```

- ▶ Calls the super class dispose method to make sure any resources used by the frame are properly freed.

All other methods used in the dispose method are the same as in the JDBC and StoredProcedure examples.

#### 4.6.13 Network print

The network print classes provide the following functions:

- ▶ Read and write iSeries spooled files
- ▶ Generate SCS data streams
- ▶ Manage print resources:
  - List, hold, and release spooled files
  - List, hold, and release output queues
  - Start and stop iSeries writer jobs
  - List and retrieve attributes of printer devices
  - List and read AFP resources

Using the Network Print classes involves the following steps:

1. Establish a connection.
2. Create a spooled file list.
3. Set the user filter.
4. Open the spooled file list.
5. Retrieve entries.
6. Close the spooled file list.
7. Close the connection.

#### 4.6.14 Print example

In this example, we use the SpooledFileList feature of the Toolbox to allow a Java program to directly access spooled files on the iSeries server (Figure 4-18).

Spooled files can be created, deleted, held, and released. Spooled files can also be filtered by user name.

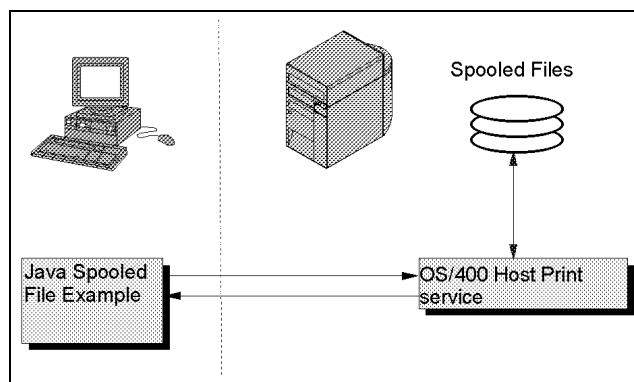


Figure 4-18 IBM Toolbox for Java Print

The Toolbox classes used are:

► **AS400(String, String, String)**

Constructor for class com.ibm.as400.access.AS400. Constructs an AS400 object for the specified system, user ID, and password.

► **SpoooledFileList(AS400)**

Constructor for class com.ibm.as400.access.SpoooledFileList. Constructs a SpoooledFileList to an iSeries server.

► **SpoooledFile()**

Constructor for class com.ibm.as400.access.SpoooledFile. Constructs a spooled file object.

The spooledFileList methods used include:

► **openSynchronously()**

Builds the list synchronously. This method does not return until the list is built completely.

► **getObject(int index)**

Returns one object from the list.

► **close()**

Closes the spooled file list.

This application was created using VisualAge for Java and the Toolbox classes. The spooled file list retrieval application allows us to retrieve a list of spooled files of the current logged on user, but any user name can be used. Figure 4-19 shows the spooled files for the \*ALL users.

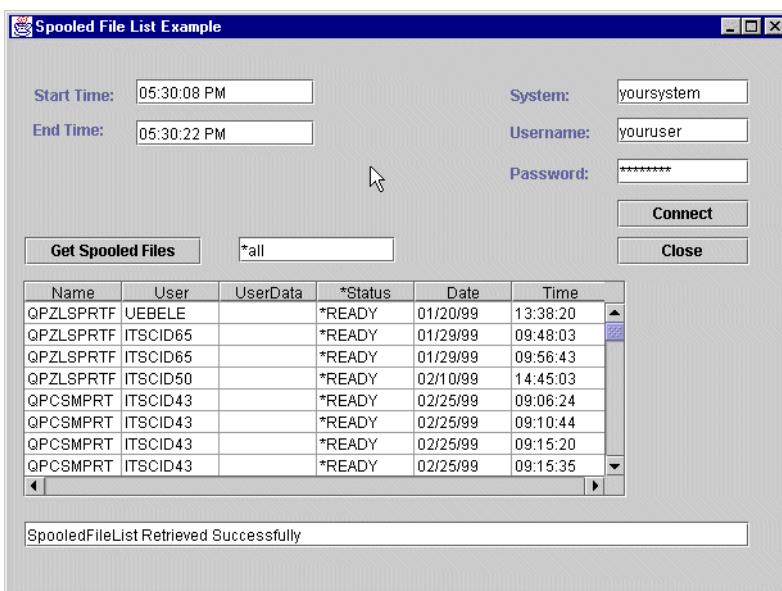


Figure 4-19 Spooled file example

### SpoooledFileListExample class

This section looks at the key methods of the SpoooledFileListExample class.

#### ***The connect method***

The connect method is called when the Connect button is clicked. String parameters representing the iSeries server name, user ID, and password are passed to the method.

---

*Example 4-35 Spooled file example connect method*

---

```
public String connect(String systemName, String userid, String password)
{
    try
    {
        as400 = new AS400(systemName, userid, password);
        as400.connectService(as400.PRINT);
    }
    catch (Exception e)
    {
        return "Exception " + e;
    }
    return "Connected";
}
```

---

This method is explained here:

```
as400 = new AS400(systemName, userid, password);
```

- ▶ Creates a new AS/400 connection object. System name, user ID, and password are passed through the constructor.

```
as400.connectService(AS400.PRINT);
```

- ▶ Connect to the iSeries Program Call and Print Service server. This is not a required call. If a service connection is needed and does not already exist, the service is connected automatically. We choose to place the connection overhead in the connect method, as opposed to connecting the first time that the user requests a spooled file.

***The formatSpooledFile method***

The formatSpooledFile method is called to format the Print objects so they can be added to the default table model of JTable. It is called by the getSpooledFilesForUser method. It is called with a SpooledFile object as input. It retrieves the attributes of the object and returns a String array that contains the formatted attributes of the object.

---

*Example 4-36 Spooled file example formatSpooledFile method*

---

```
public String[] formatSpooledFile(com.ibm.as400.access.SpooledFile theFile) throws
Exception
{
    String result[] = new String[6];
    result[0] = theFile.getStringAttribute(PrintObject.ATTR_SPOOLFILE);
    result[1] = theFile.getStringAttribute(PrintObject.ATTR_JOBUSER);
    result[2] = theFile.getStringAttribute(PrintObject.ATTR_USERDATA);
    result[3] = theFile.getStringAttribute(PrintObject.ATTR_SPLFSTATUS);
    String date = theFile.getStringAttribute(PrintObject.ATTR_DATE);
    result[4] = date.substring(3, 5) + "/" + date.substring(5, 7) + "/" + date.substring(1,
            3);
    String time = theFile.getStringAttribute(PrintObject.ATTR_TIME);
    result[5] = time.substring(0, 2) + ":" + time.substring(2, 4) + ":" + time.substring(4,
            6);
    return result;
}
```

---

### ***The getSpooledFilesForUser (String User) method***

The highlights of the getSpooledFilesForUser method include:

► **spooledFileList (as400)**

This is new. It constructs a spooled file list object using the system object. The default list shows all spooled files for the current user on the specified system.

► **list.setUserFilter (String user name)**

Specifies the user data that the spooled file must have for it to be included in the list. The value can be any specific value or the special \*ALL value. The value cannot be greater than 10 characters. The default is \*ALL.

► **list.openSynchronously()**

Builds the list synchronously. This method does not return until the list is built completely. The caller may then call the getObjects method to get an enumeration of the list.

► **list.size()**

Returns the current size of the list.

► **currentFile = (SpooledFile)list.getObject(x)**

Returns one object from the list.

► **getDefaultTableModel1().addRow(formatSpooledFile(currentFile))**

Calls formatSpooledFile to retrieve the attributes of the object and then adds it to the JTable.

► **list.close()**

Closes the list so that objects in the list can be garbage collected.

---

***Example 4-37 Spooled file example getSpooledFileForUser method***

---

```
public String getSpooledFilesForUser(String user)
{
    String res = "SpooledFileList Retrieved Successfully";
    SpooledFileList list;
    try
    {
        clearTable();
        list = new SpooledFileList(as400);
        list.setUserFilter(user.toUpperCase());
        list.openSynchronously();
        int listsize = list.size();
        SpooledFile currentFile;
        for (int x = 0; x < listsize; x++)
        {
            currentFile = (SpooledFile) list.getObject(x);
            getDefaultTableModel1().addRow(formatSpooledFile(currentFile));
        }
        if (listsize <= 0) res = "No Spooled File for User: " + user;
    } catch (Exception e)
    {
        return "An exception occurred" + e;
    }
    list.close();
    return res;
}
```

---

## 4.6.15 Integrated file system access

The integrated file system classes allow a Java program to access files in the iSeries integrated file system as a stream of bytes or a stream of characters. The integrated file system classes were created because the java.io package does not provide file redirection and other iSeries functionality.

The function that is provided by the IFSFile classes is a superset of the function provided by the file IO classes in the java.io package. All methods in java.io – FileInputStream, FileOutputStream, and RandomAccessFile – are in the integrated file system classes.

In addition to these methods, the classes contain methods to:

- ▶ Specify a file sharing mode to deny access to the file while it is in use
- ▶ Specify a file creation mode to open, create, or replace the file
- ▶ Lock a section of the file and deny access to that part of the file while it is in use
- ▶ List the contents of a directory more efficiently
- ▶ Cache the contents of a directory to improve performance by limiting calls to iSeries
- ▶ Determine the number of bytes available on the iSeries file system
- ▶ Allow a Java applet to access files in the iSeries file system
- ▶ Read and write data as text instead of as binary data
- ▶ Determine the type of the file object (logical, physical, save, and so on) when the object is in the QSYS.LIB file system

Through the integrated file system classes, the Java program can directly access stream files on the iSeries. The Java program can still use the java.io package, but the client operating system must then provide a method of redirection. For example, if the Java program is running on a Windows 95 or Windows NT operating system, the Network Drives function of IBM iSeries Client Access Express for Windows is required to redirect java.io calls to the iSeries. With the integrated file system classes, you do not need Client Access Express.

A required parameter of the integrated file system classes is the AS400 object that represents the iSeries server that contains the file.

Using the integrated file system classes causes the AS400 object to connect to the iSeries server.

### IFSJavaFile class

The IFSJavaFile class represents a file in the iSeries integrated file system and extends the java.io.File class. IFSJavaFile allows you to write files for the java.io.File interface that access iSeries integrated file systems.

IFSJavaFile makes portable interfaces that are compatible with java.io.File and uses only the errors and exceptions that java.io.File uses. IFSJavaFile uses the security manager features from java.io.File. However, unlike java.io.File, IFSJavaFile uses security features continuously.

You use IFSJavaFile with IFSFileInputStream and IFS FileOutputStream. It does not support java.io.FileInputStream and java.io.FileOutputStream.

IFSJavaFile is based on IFSFile. However, its interface is more like java.io.File than IFSFile. IFSFile is an alternative to the IFSJavaFile class.

You can get the list of files in a directory by using either the `list()` method or the `listFiles()` method:

- ▶ **listFiles() method:** Performs better because it retrieves and caches information for each file on the initial call. After that, information on each file is retrieved from the cache.

- ▶ **list()** method: Retrieves information on each file in a separate request, making it slower and more demanding of server resources.

The following code snippet accesses an iSeries called *SystemX* and writes an array of bytes to a file named *myFile.txt* in the /home directory of the iSeries server IFS:

```
IFSJavaFile file = new IFSJavaFile(new AS400("SystemX"), "/home/MyFile.txt");
try {
    IFSFileOutputStream os = new IFSFileOutputStream(file);
    byte[] data = new byte[256];
    for (int i=0; i < data.length; i++) {
        data[i] = (byte) i;
        os.write(data[i]);
    }
    os.close();
}
catch (Exception e) {
    System.err.println ("Exception: " + e.getMessage());
}
```

#### 4.6.16 Integrated file system example

In this example, we use the IFSFile class and other integrated file system classes of the IBM Toolbox for Java. This allows a Java program to interface with the OS/400 host servers to gain access to files in the iSeries integrated file system (Figure 4-20).

The Toolbox classes used in this example are:

- ▶ **AS400(String, String, String)**

Constructor for the com.ibm.as400.access.AS400 class. It constructs an AS400 object for the specified system, user ID, and password.

- ▶ **IFSFfile(AS400, String)**

Constructor for the com.ibm.as400.access.IFSFile class. It constructs an object referring to an IFS file on the iSeries server.

- ▶ **IFSFfileInputStream(AS400, IFSFile, int)**

Constructor for the com.ibm.as400.access.IFSFileInputStream class. It constructs an input stream to read the contents of the file.

The IFS methods used in this example are:

- ▶ **list()**

The method in the com.ibm.as400.access.IFSFile class. If the IFSFile object represents a directory or folder, this method returns an array of strings that holds the list of all files and directories within it.

- ▶ **getSystem()**

The method in the com.ibm.as400.access.IFSFile class. It returns the AS400 object from which this IFSFile was created.

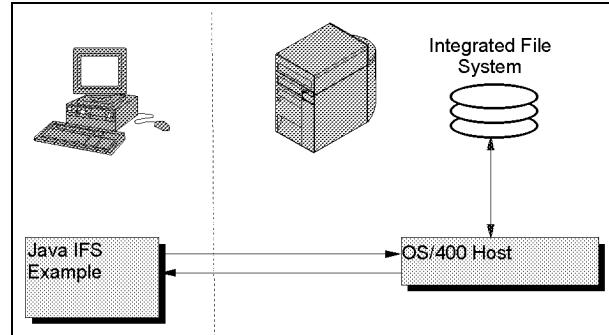


Figure 4-20 IBM Toolbox for Java IFS

► **getName()**

The method in the com.ibm.as400.access.IFSFile class. It returns a string with the name of the IFSFile.

► **isDirectory() and isFile()**

The methods in the com.ibm.as400.access.IFSFile class. It return booleans that indicate whether the IFSFile object represents a file or directory.

► **length()**

The method in the com.ibm.as400.access.IFSFile class. It returns the length (in bytes) of the file.

► **lastModified()**

The method in the com.ibm.as400.access.IFSFile class. It returns the date that the file was last modified (as a long).

► **available()**

The method in the com.ibm.as400.access.IFSFileInputStream class. It returns the number of available bytes in the file.

► **read(byte[])**

The method in the com.ibm.as400.access.IFSFileInputStream class. It reads the number of bytes available and stores in the byte array.

► **close()**

The method in the com.ibm.as400.access.IFSFileInputStream class. It closes the input stream.

This application was built using VisualAge for Java and the Toolbox. In this example, we use the integrated file system classes of the IBM Toolbox to allow a Java program to retrieve a list of files from the iSeries integrated file system. Figure 4-21 shows the files stored in the IFS for the path entered in the text box.

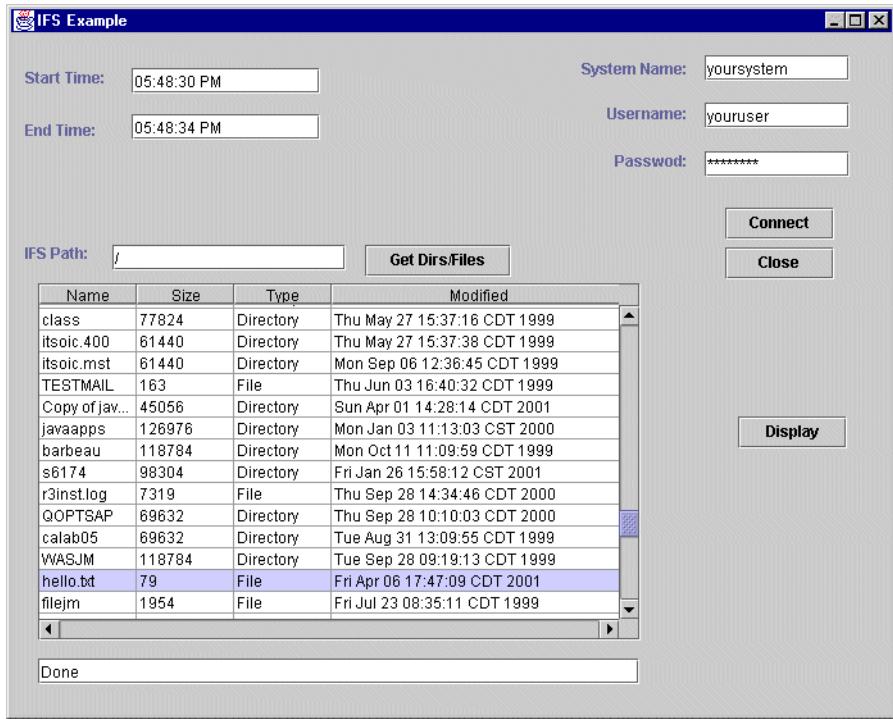


Figure 4-21 Integrated file system example (directories/files)

If a text file is selected from the list of files, its contents are displayed as shown in Figure 4-22.

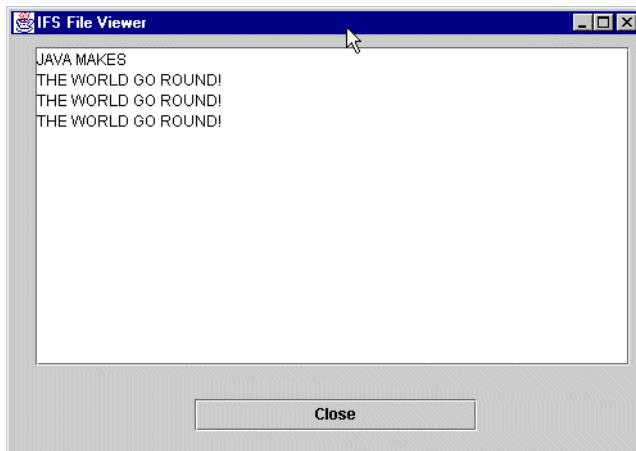


Figure 4-22 Integrated file system example (file viewer)

To build this application, the following steps are required:

1. Establish a connection.
2. Set the IFS Path to view.
3. Create an IFSFile object.
4. Retrieve the list of files.
5. Open an IFSFileInputStream.
6. Read the file contents.
7. Close the connection.

## **IFSExample class**

This section examines the key methods of the IFSExample class.

### **The connect method**

The connect method is called when the Connect button is clicked. String parameters representing the iSeries server name, user ID, and password are passed to the method.

---

#### *Example 4-38 Integrated file system example connect method*

---

```
public void connect(String systemName, String userid, String password) throws Exception
{
    getStatus().setText("Connecting....");
    // as400 is declared elsewhere in the class
    as400 = new com.ibm.as400.access.AS400(systemName, userid, password);
    as400.connectService(com.ibm.as400.access.AS400.FILE);
    getStatus().setText("Connected to AS400");
    return;
}
```

---

The highlights of the connect method include:

```
as400 = new com.ibm.as400.access.AS400(systemName, userid, password);
```

- ▶ Creates a new AS/400 connection object. The system name, user ID, and password are passed through the constructor.
- as400.connectService(AS400.com.ibm.as400.access.AS400.FILE);
- ▶ Connects to the iSeries host file server. This is not a required call. If a service connection is needed and does not already exist, the service is connected automatically. We chose to place the connection overhead in the connect method as opposed to connecting the first time the user requests to access a file.

### **The populateTable method**

The populateTable method is called when the Get Dirs/Files button is clicked. A string parameter representing the IFS path is passed to the method.

---

#### *Example 4-39 Integrated file system example populateTable method*

---

```
public void populateTable(String IFSPath)
{
    com.ibm.as400.access.IFSFile aFile;
    String[] files;
    Object rowKey;
    getStatus().setText("Retrieving...");
    clearTable();
    try
    {
        aFile = new com.ibm.as400.access.IFSFile(as400, IFSPath);
        files = aFile.list();
        for (int i=0; i<files.length; i++)
        {
            aFile = new com.ibm.as400.access.IFSFile(as400, IFSPath, files[i]);
            rowKey = aFile;
            String[] rowData = new String[4];
            rowData[0] = files[i];
            rowData[1] = String.valueOf(aFile.length());
            if (aFile.isDirectory())

```

```

        {
            rowData[2] = "Directory";
        }
        else
        {
            rowData[2] = "File";
        }
        rowData[3] = new java.util.Date(aFile.lastModified()).toString();
        getDefaultTableModel1.addRow(rowData);
    }
}
catch (java.io.IOException ex)
{
    System.out.println("Error Receiving Files: "+ex);
}
getStatus().setText("Done.");
return;
}

```

---

The populateTable method highlights include:

```
aFile = new com.ibm.as400.access.IFSFile(as400, IFSPath);
```

- ▶ Creates a new IFSFile object using the AS400 object and the IFS path as parameters.

```
files = aFile.list();
```

- ▶ Uses the IFSFile list method to return an array of strings that holds a list of files and directories held in the IFSFile object.
- ```
for (int i=0; i<files.length; i++) {
    aFile = new com.ibm.as400.access.IFSFile(as400, IFSPath, files[i]);
    .
    .
}
```
- ▶ Loops through the list of file objects stored in the string array named *files* and builds an object array that contains the name of the file, the size, the type, and the last modified date.

```
getDefaultTableModel1().addRow(rowData);
```

- ▶ Adds a new entry to the JTable for the file object.

### ***The readFile() method***

The readFile() method reads the contents of a file as a stream of bytes and stores them in a byte array.

---

#### *Example 4-40 Integrated file system example readFile method*

---

```
protected void readFile()
{
    byte[] data = null;

    // Determine if the file extension is .txt
    String name = _file.getName();
    int index = name.lastIndexOf(".");
    if (index == -1)
    {
        getFileContents().setText("Error: Only .txt files can be viewed.");
        return;
    }
```

```

        }
        if (!(name.substring(index + 1).toUpperCase()).equals("TXT"))
        {
            getFileContents().setText("Error: Only .txt files can be viewed.");
            return;
        }
        try
        {
            IFSFileInputStream in = new IFSFileInputStream(_file.getSystem(), _file,
                IFSFileInputStream.SHARE_ALL);
            int len = in.available();
            data = new byte[len];
            in.read(data);
            in.close();
        }
        catch (Exception ex)
        {
            System.err.println("Error reading file: " + ex);
        }
        String t = new String(data);
        getFileContents().setText(t);
    }

```

---

The highlights of the Readfile() method are listed here:

```
IJSFileInputStream in = new IJSFileInputStream(_file.getSystem(), _file,
    IJSFileInputStream.SHARE_ALL);
```

- ▶ Creates a new IJSFileInputStream object that is used to read the contents of a file.

```
int len = in.available();
```

- ▶ Uses the IJSFileInputStream available method to determine the number of bytes contained in the file.

```
data = new byte[len];
```

- ▶ Allocates a byte array to hold the data of the size returned previously.

```
in.read(data);
```

- ▶ Uses the IJSFileInputStream read method to read the stream of bytes into the byte array.

#### 4.6.17 Additional access classes

This section describes and demonstrates some of the new or enhanced classes in the access package. All sample applications were written using VisualAge for Java 3.5 and Modification 4 of the IBM Toolbox for Java loaded in the workspace.

##### **FTP classes**

The FTP class provides a programmable interface to FTP functions. You no longer have to use Runtime.exec() or java.lang.Runtime.exec() or tell your users to run FTP commands in a separate application. That is, you can program FTP functions directly into your application. From within your program, you can:

- ▶ Connect to an FTP server
- ▶ Send commands to the server

- ▶ List the files in a directory
- ▶ Get files from the server
- ▶ Put files to the server

For example, with the FTP class, you can copy a set of files from a directory on a server:

```
FTP client = new FTP("myServer", "myUID", "myPWD");
client.cd("/myDir");
client.setDataTransferType(FTP.BINARY);
String [] entries = client.ls();
for (int i = 0; i < entries.length; i++)
{
    System.out.println("Copying " + entries[i]);
    try
    {
        client.get(entries[i], "c:\\\\ftptest\\\\" + entries[i]);
    }
    catch (Exception e)
    {
        System.out.println(" copy failed, likely this is a directory");
    }
}
client.disconnect();
```

FTP is a generic interface that works with many different FTP servers. Therefore, it is up to the programmer to match the semantics of the server.

### **FTP subclass**

While the FTP class is a generic FTP interface, the AS400FTP subclass is written specifically for the FTP server on the iSeries server. That is, it understands the semantics of the FTP server on the iSeries, so the programmer doesn't have to. For example, this class understands the various steps needed to transfer an iSeries save file to the iSeries and performs these steps automatically.

AS400FTP also ties into the security facilities of the IBM Toolbox for Java. As with other IBM Toolbox for Java classes, AS400FTP depends on the AS400 object for system name, user ID, and password.

The following example sends (puts) a save file to the iSeries server. Note the application does not set data transfer type to binary or use the Toolbox CommandCall to create the save file. Since the extension is .savf, the AS400FTP class detects the file to put is a save file so it does these steps automatically.

```
AS400 system = new AS400();
AS400FTP ftp = new AS400FTP(system);
ftp.put("myData.savf", "/QSYS.LIB/MYLIB.LIB/MYDATA.SAVF");
```

### **JavaApplicationCall class**

The JavaApplicationCall class provides you with the ability to easily run a Java program residing on the iSeries server from a client with the Java virtual machine for iSeries.

After establishing a connection to the iSeries server from the client, the JavaApplicationCall class lets you:

- ▶ Set the CLASSPATH environment variable on the iSeries with the setClassPath() method
- ▶ Define your program's parameters with the setParameters() method
- ▶ Run the program with the run() method

- ▶ Send input from the client to the Java program. The Java program reads the input via standard input, which is set with the `sendStandardInString()` method. You can redirect standard output and standard error from the Java program to the client via the `getStandardOutString()` and `getStandardErrorString()` methods.

You call the `JavaApplicationCall` class from your Java program. However, IBM Toolbox for Java also provides utilities to call iSeries Java programs. These utilities are complete Java programs you can run from your workstation. See the `RunJavaApplication` class for more information.

### ***JavaApplicationCall example***

Supposed the Java class `HelloWorld` resides in directory `/javatest` on the iSeries server. The following code snippet calls this program and receives program output written to standard out.

```
import com.ibm.as400.access.*;
public class TestJavaApplicationCall implements Runnable {
    JavaApplicationCall jaCall;

    public static void main(String[] args)
    {
        TestJavaApplicationCall test = new TestJavaApplicationCall();
        test.javaPgmCall(args);
    }

    void javaPgmCall(String[] args)
    {
        try
        {
            AS400 as400 = new AS400();
            jaCall = new JavaApplicationCall(as400);
            jaCall.setJavaApplication("HelloWorld");
            jaCall.setClassPath("/javatest");
            Thread outputThread = new Thread(this);
            outputThread.start();
            if (jaCall.run() != true)
            {
                AS400Message[] messageList = jaCall.getMessageList();
                for (int msg = 0; msg < messageList.length; msg++)
                    System.out.println(messageList[msg].toString());
            }
        }
        catch (Exception e) { e.printStackTrace(); }
        System.exit(0);
    }
    public void run()
    {
        while(true)
        {
            String s = jaCall.getStandardOutString();
            if (s != null)
                System.out.println(s);
            try { Thread.sleep(100); } catch (Exception e) {}
        }
    }
}
```

## 4.7 Introduction to GUI component classes

The IBM Toolbox for Java provides a set of graphic user interface (GUI) classes in the `vaccess` package. With the GUI classes, you can visually represent your iSeries data and resources. This section covers some of the GUI classes.

Java programs that use the IBM Toolbox for Java GUI classes also need Sun's JDK Swing 1.0.1 or later support. Swing is available with Sun's JFC (Java Foundation Classes) 1.1. For more information about Swing, visit the Web site at:  
<http://www.javasoft.com/products/jfc/index.html>

VisualAge for Java 3.5 provides this support so you can use these classes in the Visual Composition Editor. The IBM Toolbox for Java provides GUI classes for the following resources:

- ▶ iSeries panes
- ▶ Java Database Connectivity (JDBC)
- ▶ Data queues
- ▶ Command call
- ▶ Error events
- ▶ Jobs
- ▶ Messages
- ▶ Network print
- ▶ Program call
- ▶ Record-level access
- ▶ Users and groups
- ▶ System status
- ▶ System values
- ▶ Resource list

### 4.7.1 Overview of the GUI classes

This section presents an overview of each of the GUI classes. Basic programming examples are provided in the *IBM Toolbox for Java API Users Guide* that is available with the IBM Toolbox for Java. We cover examples that use the GUI classes and VisualAge for Java 3.5 to produce iSeries client/server applications.

#### iSeries panes

iSeries panes are graphical user interface classes that present and allow manipulation of one or more iSeries resource. The behavior of each iSeries resource varies depending on the type of resource. All panes extend the Java Component class. As a result, they can be added to any Swing JFrame, Window, or Container.

The following iSeries panes are available:

- ▶ **AS400ListPane** presents a list of iSeries resources and allows selection of one or more resources (Figure 4-23 on page 220).

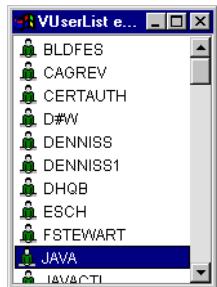


Figure 4-23 AS400ListPane

- ▶ **AS400DetailsPane** presents a list of iSeries resources in a table where each row displays various details about a single resource. The table allows the selection of one or more resources (Figure 4-24).



Figure 4-24 AS400DetailsPane

- ▶ **AS400TreePane** presents a tree hierarchy of iSeries resources and allows the selection of one or more resources (Figure 4-25).

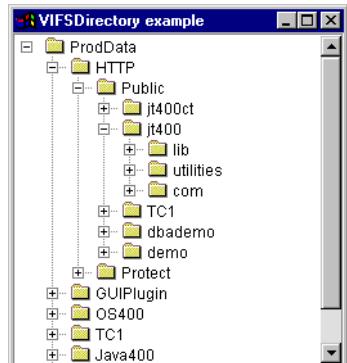


Figure 4-25 AS400TreePane

- ▶ **AS400ExplorerPane** combines an AS400TreePane and AS400DetailsPane so that the resource selected in the tree is presented in the details (Figure 4-26).



Figure 4-26 AS400ExplorerPane

## JDBC

Java Database Connectivity (JDBC) graphical user interface classes allow a Java program to present various views and controls for accessing a database using Structured Query Language (SQL) statements and queries. The following classes are available:

- ▶ **SQLStatementButton**: A button that issues an SQL statement when clicked.
- ▶ **SQLStatementMenuItem**: A menu item that issues an SQL statement when selected.
- ▶ **SQLStatementDocument**: A document that can be used with any Java Foundation Classes (JFC) graphical text component to issue an SQL statement.
- ▶ **SQLResultSetFormPane**: Presents the results of an SQL query in a form.
- ▶ **SQLResultSetTablePane**: Presents the results of an SQL query in a table.
- ▶ **SQLResultSetTableModel**: Manages the results of an SQL query in a table.
- ▶ **SQLQueryBuilderPane**: Presents an interactive tool for dynamically building SQL statements.

### ***SQL connections***

An SQLConnection object represents a connection to a database using JDBC. The SQLConnection object is used with all of the JDBC graphical user interface classes. To use an SQLConnection, set the URL property using the constructor or setURL() method. This identifies the database to which the connection is made. Other optional properties can be set:

- ▶ Use **setProperties()** to specify a set of JDBC connection properties.
- ▶ Use **setUserName()** to specify the user name for the connection.
- ▶ Use **setPassword()** to specify the password for the connection.

The actual connection to the database is not made when the SQLConnection object is created. Instead, it is made when the getConnection() method is called. This method is normally called automatically by the JDBC graphical user interface classes, but it can be called at any time to control when the connection is made.

An SQLConnection object can be used for more than one JDBC graphical user interface component. All such classes use the same connection, which can improve performance and resource usage. Alternately, each JDBC graphical user interface component can use a different SQL object. It is sometimes necessary to use separate connections so that SQL statements are issued in different transactions. When the connection is no longer needed, close the SQLConnection object using the close() method. This frees up JDBC resources on both the client and server.

### **Command call**

The command call GUI components allow a Java program to present a button or menu item that calls a non-interactive iSeries command. A CommandCallButton object represents a button that calls an iSeries command when clicked. The CommandCallButton class extends the JFC JButton class so that all buttons have a consistent appearance and behavior.

Similarly, a CommandCallMenuItem object represents a menu item that calls an iSeries command when selected. The CommandCallMenuItem class extends the JFC JMenuItem class so that all menu items also have a consistent appearance and behavior. To use a command call graphical user interface component, set both the system and command properties. These properties can be set using a constructor or through the setSystem() and setCommand() methods.

### **Data queues**

The data queue graphical components allow a Java program to use any JFC graphical text component to read or write to an iSeries data queue. The DataQueueDocument and KeyedDataQueueDocument classes are implementations of the JFC Document interface. These classes can be used directly with any JFC graphical text component.

Several text components, such as single line fields (JTextField) and multiple line text areas (JTextArea), are available in JFC. Data queue documents associate the contents of a text component with an iSeries data queue.

**Note:** A *text component* is a graphical component used to display text that the user can optionally edit.

The Java program can read and write between the text component and data queue at any time. Use DataQueueDocument for sequential data queues, and use KeyedDataQueueDocument for keyed data queues. To use a DataQueueDocument, set both the system and path properties. These properties can be set using a constructor or through the setSystem() and setPath() methods. The DataQueueDocument object is then “plugged” into the text component, usually using the text component's constructor or setDocument() method. KeyedDataQueueDocuments work the same way.

## Error events

In most cases, the IBM Toolbox for Java GUI components fire error events instead of throw exceptions. An *error event* is a wrapper around an exception that is thrown by an internal component. You can provide an error listener that handles all error events that are fired by a particular graphical user interface component.

Whenever an exception is thrown, the listener is called, and it can provide appropriate error reporting. By default, no action takes place when error events are fired. The IBM Toolbox for Java provides a graphical user interface component called ErrorDialogAdapter, which automatically displays a dialog to the user whenever an error event is fired.

## Jobs

The jobs GUI classes allow a Java program to present lists of iSeries jobs and job log messages in a graphical user interface. The following classes are available:

- ▶ **VJobList object:** A resource that represents a list of iSeries jobs for use in iSeries panes.
- ▶ **VJob object:** A resource that represents the list of messages in a job log for use in iSeries panes.

You can use iSeries panes, VJobList objects, and VJob objects together to present many views of a job list or job log.

The VJobList example shown in Figure 4-27 presents an AS400ExplorerPane filled with a list of jobs. The list shows jobs on the system that have the same job name.

The screenshot shows a Windows-style application window titled "Job list example". On the left, there is a tree view under the "Job list" node, which has multiple entries labeled "QZDASOINIT". On the right, there is a table with the following columns: Job name, User, Job nu..., Subsystem, and Type. The table contains 20 rows, each corresponding to one of the "QZDASOINIT" entries in the tree view. The data in the table is as follows:

Job name	User	Job nu...	Subsystem	Type
QZDASOINIT	QUSER	014994	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	015026	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016098	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016132	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016133	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016144	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016234	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016299	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016300	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016301	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016302	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016303	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016304	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016305	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016314	/QSYS/QSERVER	Batch

Figure 4-27 VJobList graphical user interface component

## Messages

The messages GUI components allow a Java program to present lists of iSeries messages in a graphical user interface. The following components are available:

- ▶ **Message list object** is a resource that represents a list of messages for use in iSeries panes. This is for message lists generated by command or program calls.
- ▶ **Message queues object** is a resource that represents the messages in an iSeries message queue for use in iSeries panes.

iSeries panes are graphical user interface components that present and allow manipulation of one or more iSeries resources. VMessageList and VMessageQueue objects are resources that represent lists of iSeries messages in iSeries panes.

You can use iSeries pane, VMessageList, and VMessageQueue objects together to present many views of a message list and to allow the user to select and perform operations on messages.

## Network print

The GUI print components allow a Java program to present lists of iSeries print resources in a GUI. The following components are available:

- ▶ **VPrinters object** is a resource that represents a list of printers for use in iSeries panes.
- ▶ **VPrinter object** is a resource that represents a printer and its spooled files for use in iSeries panes.
- ▶ **VPrinterOutput object** is a resource that represents a list of spooled files for use in iSeries panes.
- ▶ **SpooledFileViewer object** is a resource that visually represents spooled files. iSeries panes are graphical user interface components that present and allow manipulation of one or more iSeries resources.

VPrinters, VPrinter, and VPrinterOutput objects are resources that represent lists of iSeries print resources in iSeries panes.

iSeries pane, VPrinters, VPrinter, and VPrinterOutput objects can be used together to present many views of print resources and to allow the user to select and perform operations on them.

## Program call

The program call GUI components allow a Java program to present a button or menu item that calls an iSeries program. Input, output, and input/output parameters can be specified using ProgramParameter objects. When the program runs, the output and input/output parameters contain data returned by the iSeries program.

A ProgramCallButton object represents a button that calls an iSeries program when it is clicked. The ProgramCallButton class extends the JFC JButton class so that all buttons have a consistent appearance and behavior.

Similarly, a ProgramCallMenuItem object represents a menu item that calls an iSeries program when selected. The ProgramCallMenuItem class extends the JFC JMenuItem class so that all menu items also have a consistent appearance and behavior.

To use a program call graphical user interface component, set both the system and program properties. Set these properties by using a constructor or through the setSystem() and setProgram() methods.

## Record-level access

The record-level access GUI components allow a Java program to present various views of iSeries files. The following components are available:

- ▶ **RecordListFormPane** presents a list of records from an iSeries file in a form.
- ▶ **RecordListTablePane** presents a list of records from an iSeries file in a table.
- ▶ **RecordListTableModel** manages the list of records from an iSeries file for a table.

## Keyed access

You can use the record-level access GUI components with keyed access to an iSeries file. Keyed access means that the Java program can access the records of a file by specifying a key.

Keyed access works the same for each record-level access graphical user interface component. Use `setKeyed()` to specify keyed access instead of sequential access. Specify a key using the constructor or the `setKey()` method.

By default, only records whose keys are equal to the specified key are displayed. To change this, specify the `searchType` property using the constructor or `setSearchType()` method. The possible choices are:

- ▶ **KEY\_EQ**: Display records whose keys are equal to the specified key.
- ▶ **KEY\_GE**: Display records whose keys are greater than or equal to the specified key.
- ▶ **KEY\_GT**: Display records whose keys are greater than the specified key.
- ▶ **KEY\_LE**: Display records whose keys are less than or equal to the specified key.
- ▶ **KEY\_LT**: Display records whose keys are less than the specified key.

## Users and groups

The users and groups graphical user interface components allow you to present lists of iSeries users and groups through the `VUser` class.

The following components are available:

- ▶ **iSeries panes** are GUI components that present and allow manipulation of one or more iSeries resources.
- ▶ **VUserList** object is a resource that represents a list of iSeries users and groups for use in iSeries panes.
- ▶ **VUserAndGroup** object is a resource for use in iSeries panes that represents groups of iSeries users. It allows a Java program to list all users, list all groups, or list users who are not in groups.

iSeries panes and `VUserList` objects can be used together to present many views of the list. They can also be used to allow the user to select users and groups. Figure 4-28 shows the `VUserList` graphical user interface component.

## Integrated file system

The integrated file system GUI components allow a Java program to present directories and files in the iSeries integrated file system in a GUI.

The following components are available:

- ▶ **IFSFileDialog** presents a dialog that allows the user to choose a directory and select a file by navigating through the directory hierarchy.

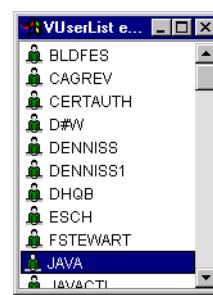


Figure 4-28 *VUserList* GUI component

- ▶ **VIFSDirectory** is a resource that represents a directory in the integrated file system for use in iSeries panes.
- ▶ **IFSTextFileDocument** represents a text file for use in any JFC graphical text component.

To use the integrated file system GUI components, set both the system and the path or directory properties. These properties can be set using a constructor or through the `setDirectory()` (for `IFSFileDialog`) or `setSystem()` and `setPath()` methods (for `VIFSDirectory` and `IFSTextFileDocument`).

You should set the path to something other than "/QSYS.LIB" because this directory is usually large, and downloading its contents can take a long time.

## **ResourceList classes**

Use the `ResourceListPane` and `ResourceListDetailsPane` classes to present a resource list in a graphical user interface (GUI).

- ▶ **ResourceListPane** displays the contents of the resource list in a graphical `javax.swing.JList`. Every item displayed in the list represents a resource object from the resource list.
- ▶ **ResourceListDetailsPane** displays the contents of the resource list in a graphical `javax.swing.JTable`. Every row in the table represents a resource object from the resource list.

The table columns for a `ResourceListDetailsPane` are specified as an array of `column` attribute IDs. The table contains a column for each element of the array and a row for each resource object. Pop-up menus are enabled by default for both `ResourceListPane` and `ResourceListDetailsPane`.

Most errors are reported as `com.ibm.as400.vaccess.ErrorEvents` rather than thrown exceptions. Listen for `ErrorEvents` to diagnose and recover from error conditions.

## **System status**

The System status GUI components allow you to create GUIs by using the existing iSeries Panes. You also have the option to create your own GUIs using the JFC. The `VSystemStatus` object represents a system status on the iSeries. The `VSystemPool` object represents a system pool on the iSeries. The `VSystemStatusPane` represents a visual pane that displays the system status information.

The `VSystemStatus` class allows you to get information about the status of an iSeries session within a GUI environment. Some of the possible pieces of information you can get are listed here:

- ▶ **getSystem()** method: Returns the iSeries server where the system status information is contained
- ▶ **getText()** method: Returns the description text
- ▶ **setSystem()** method: Sets the iSeries where the system status information is located

In addition to the methods mentioned above, you can also access and change system pool information in a graphic interface.

You use `VSystemStatus` with `VSystemStatusPane`. `VSystemStatusPane` is the visual display pane where information is shown for both system status and system pool.

## System values

The system value GUI components allow a Java program to create GUIs by using the existing iSeries panes or by creating your own panes using the JFC. The VSystemValueList object represents a system value list on the iSeries. To use the System Value GUI component, set the system name with a constructor or through the setSystem() method.

### 4.7.2 JDBC examples

This section explains how to build iSeries client/server applications using the IBM Toolbox for Java GUI JDBC classes and VisualAge for Java 3.5.

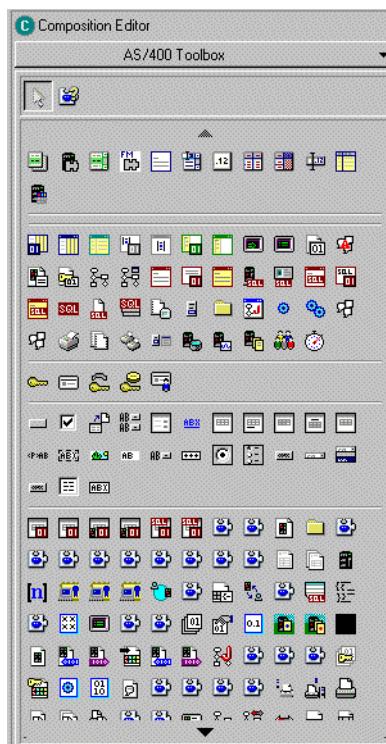


Figure 4-29 IBM Toolbox for Java classes in the VCE

## Using the IBM Toolbox for Java classes in the VCE

The IBM Toolbox for Java classes are available in the VisualAge for Java Visual Composition Editor. To use the Toolbox classes in the VCE, select the IBM Toolbox for Java from the palette pulldown choice box. Hover help is provided so you can move the mouse pointer over the component to display its name.

Figure 4-29 shows the IBM Toolbox classes as they appear in the VisualAge for Java Visual Composition Editor.

## SQLResultSetTablePane

The SQLResultSetTablePane presents the results of an SQL query in a table. In this example, we demonstrate building an iSeries client/server application using the SQLResultSetTablePane. We also use an SQLConnection component to provide connectivity to the iSeries server and an ErrorDialogAdapter to handle error conditions. All of these classes are available with the IBM Toolbox for Java. This example demonstrates building an iSeries client/server application without writing code. The GUI classes do all the work.

## SQLResultSetTablePane application

The SQLResultSetTablePane application allows the end user to enter SQL statements that run against a DB2 UDB for iSeries database. The results are displayed in an SQLResultSetTable Pane. Figure 4-30 shows the completed application.

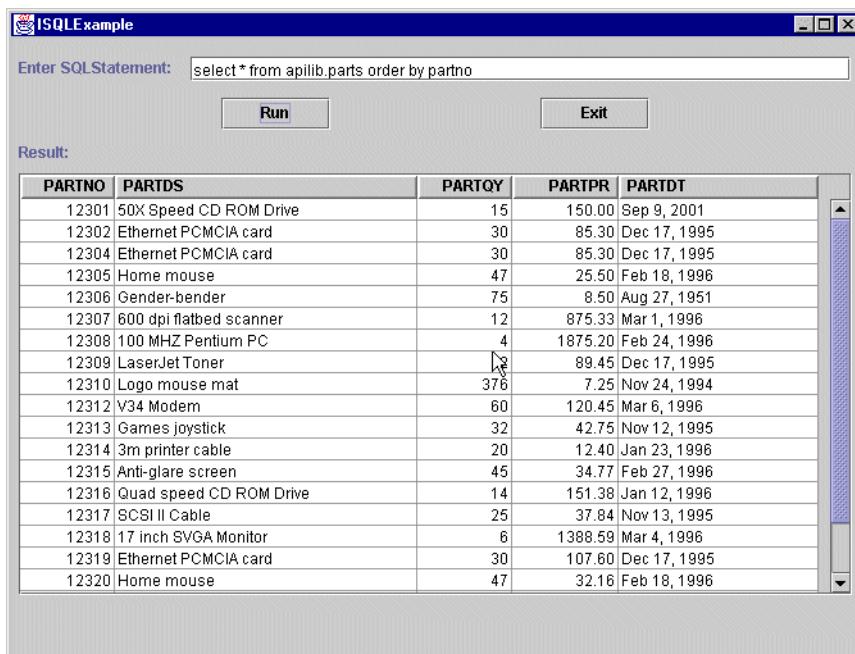


Figure 4-30 SQLResultSetTablePane example

### Error handling

We use an ErrorDialogAdapter as a listener to handle and display error conditions. In this case, we entered an SQL statement that could not be executed on the iSeries server. The resulting error condition is displayed in a dialog box, which is shown in Figure 4-31.

### Building the application

This section covers building this application using VisualAge for Java 3.5. We use the Visual Composition Editor (VCE) to construct the application from Toolbox classes.

Figure 4-32 on page 228 shows the application in the VisualAge for Java Visual Composition Editor. We use one visual and four non-visual classes to build the application. The visual component is the SQLResultSetTablePane that we place on the frame. It displays the results of an SQL statement entered in the JTextField at the top of the Frame. The DriverManager class is used to load the proper JDBC driver. The SQLConnection class is used to connect to the iSeries server. The showException method is used to display any application-defined messages. The ErrorDialogAdapter displays any errors generated by the SQLResultSetTablePane.

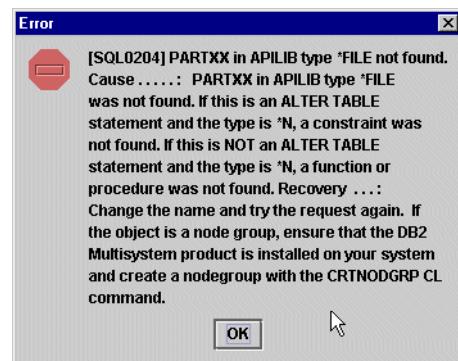


Figure 4-31 ErrorDialogAdapter dialog box

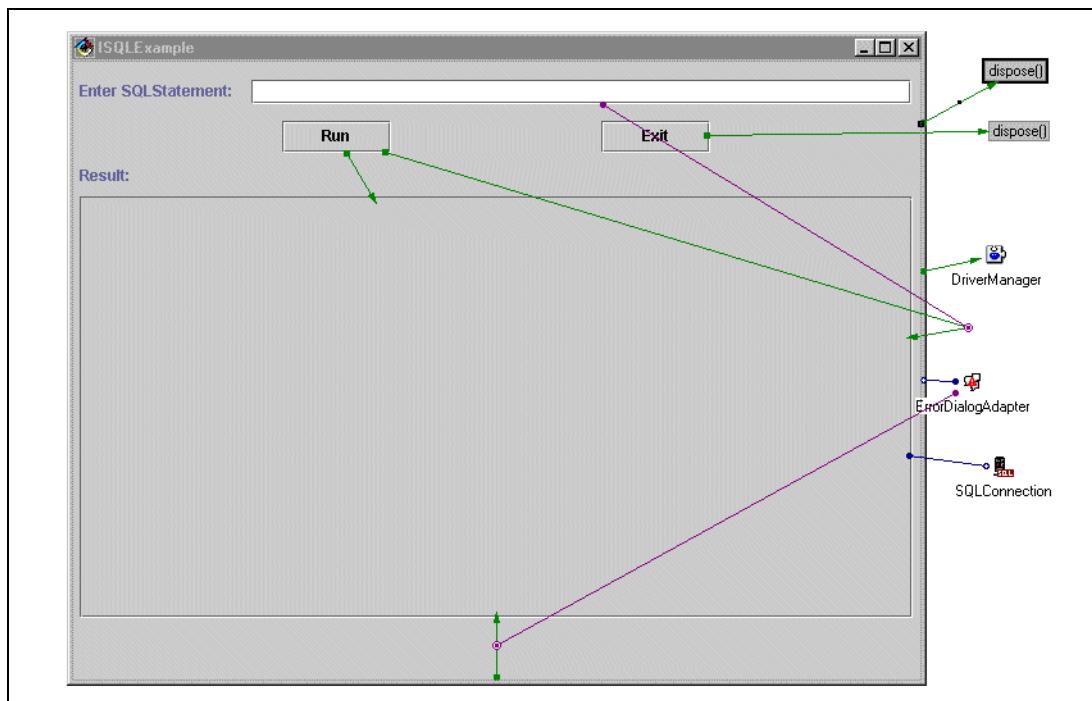


Figure 4-32 Building the application with the VisualAge for Java VCE

### **Registering the driver manager**

All JDBC GUI classes communicate with the database using a JDBC driver. The JDBC driver must be registered with the JDBC driver manager for any of these classes to work. The following code example registers the IBM Toolbox for Java JDBC driver:

```
// Register the JDBC driver
DriverManager.registerDriver(new com.ibm.as400.access.AS400JDBCDriver());
```

In this example, we drop a DriverManager object, from the java.sql package, outside the frame and use the windowOpened event of the frame to register the proper JDBC driver with the DriverManager object. Figure 4-33 shows the connection that is made.

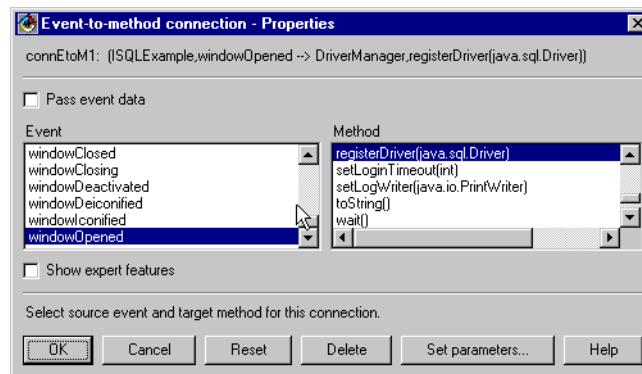


Figure 4-33 JDBC registerDriver

In the VCE, we click the Set Parameters button. Then we enter the name of the IBM Toolbox for Java JDBC driver as the name of the driver that we want to use as shown in Figure 4-34.

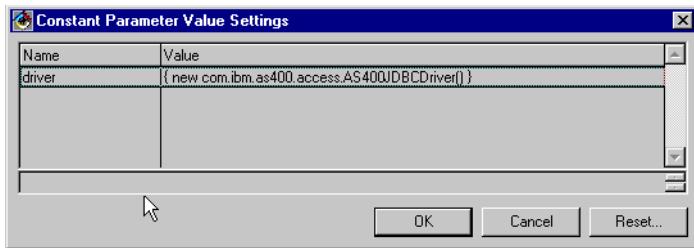


Figure 4-34 Registering the IBM Toolbox JDBC driver

We use the Toolbox SQLConnection class to handle the connection to the iSeries server. It provides methods that allow you to obtain and set its properties. Figure 4-35 shows the SQLConnection properties.

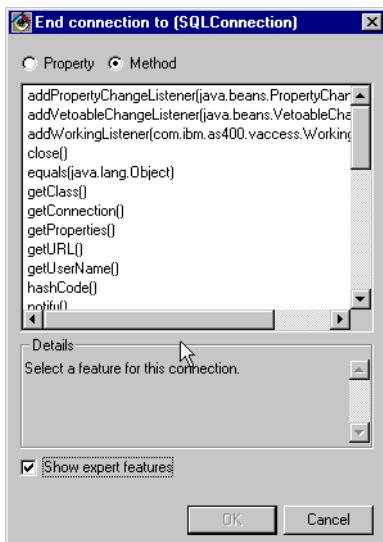


Figure 4-35 SQLConnection methods

As shown in Figure 4-36, we set the URL property in the VCE. We do not set the name of the iSeries server, the user ID, or the password. The first time we try to run a SQL statement, we are prompted for this information.

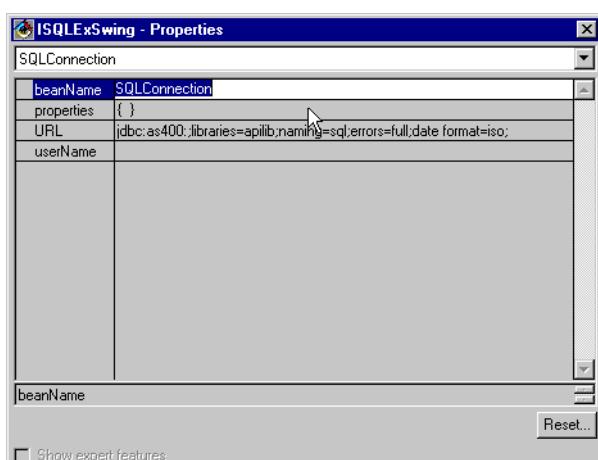


Figure 4-36 Setting the SQLConnection URL property

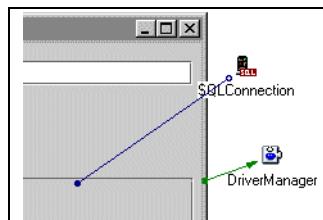


Figure 4-37 Setting the SQLResultSetTablePane connection property

The SQLResultSetTablePane is placed on the frame. It is used to display the results of the query. The connection property of the SQLResultSetTable is set to the SQLConnection component that we created earlier. This is shown in Figure 4-37.

The Run button is used to initiate the running of the query and display the results. It causes the setQuery method of the SQLResultSetTablePane to be executed with the JTextField supplying the input parameter. The load method causes the query to run and fill the SQLResultSetTablePane with the rows from the database. Figure 4-38 shows the Run button events.

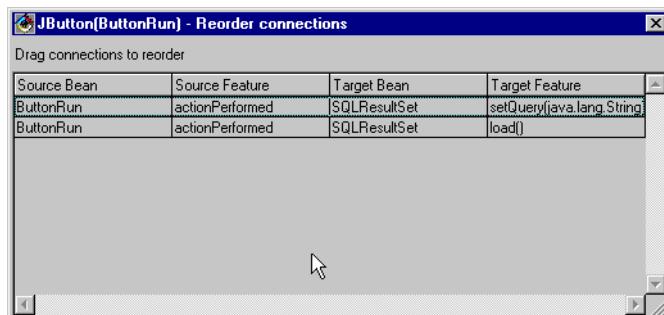


Figure 4-38 Run button events

The final component to add is the ErrorDialogAdapter to handle error conditions for our application. As shown in Figure 4-39, we use the windowOpened event to add an ErrorListener. We pass in the ErrorDialogAdapter as a parameter to the ErrorListener.

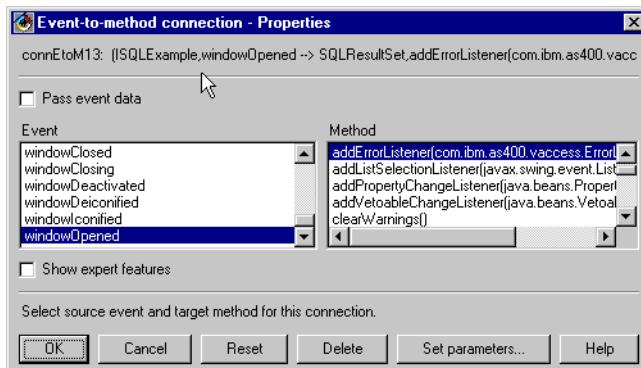


Figure 4-39 Adding an ErrorListener

This completes the application. We have created an iSeries client/server application that allows us to use JDBC to execute SQL statements against DB2 UDB for iSeries databases. We use a listener to handle and display any error conditions that occur. We did this all without writing a single line of Java code.

## SQLQueryBuilderPane

The SQLQueryBuilderPane presents an interactive tool for dynamically building SQL queries. You can use the SQLQueryBuilderPane to build and execute SQL statements against a DB2 UDB for iSeries database. It allows you to select the table that you want to use. It retrieves the table columns and displays them. You can choose which columns you want, select criteria for

the Where clause, and specify how to group or order the information. As you build the SQL statement, you can use the Summary tab to display the statement. When you are satisfied with the statement, you can execute it. The example shown in Figure 4-40 displays the results in a SQLResultSetTablePane.

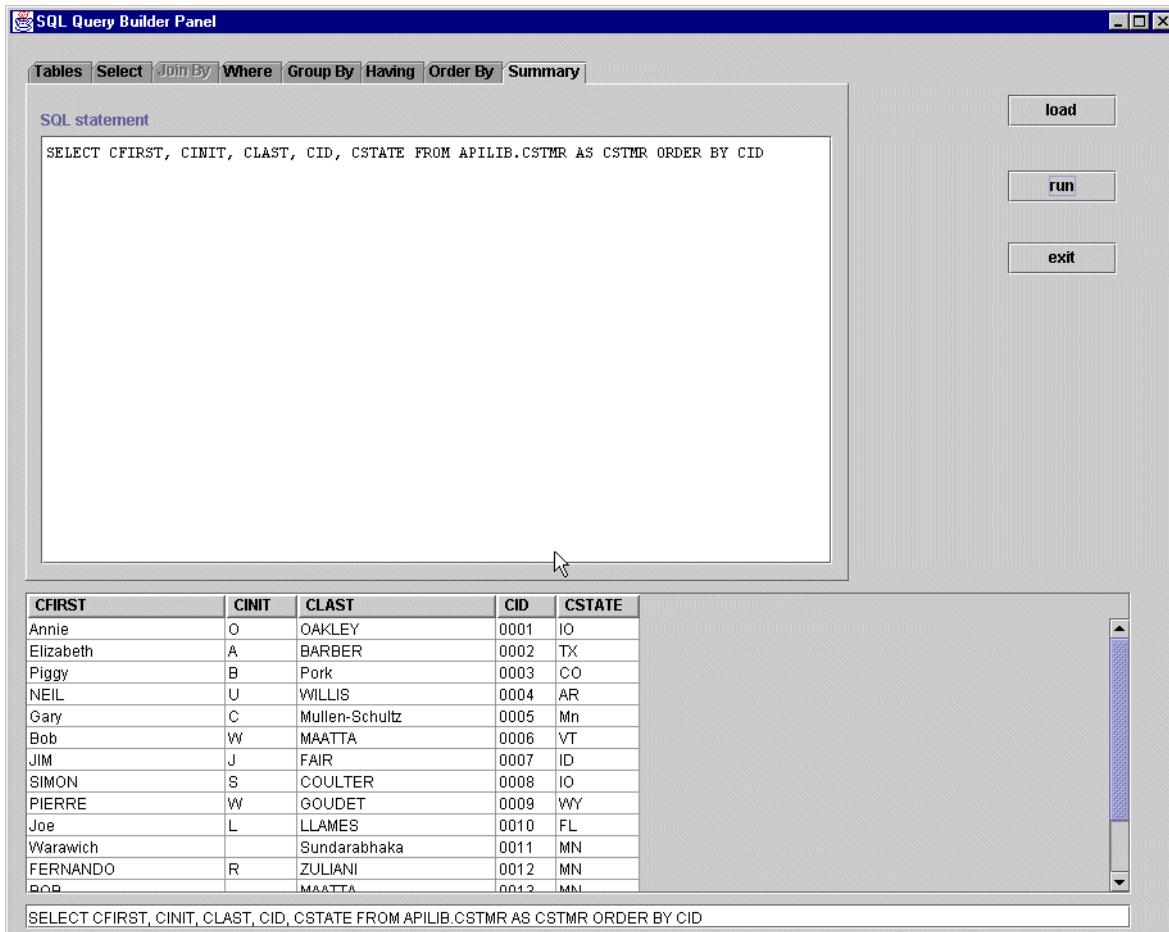


Figure 4-40 SQLQueryBuilderPane example

We use an `ErrorDialogAdapter` to display any error conditions. The `SQLQueryBuilderPane` allows you to modify the SQL statement that you are building. For example, if you enter an invalid column name, the dialog box shown in Figure 4-41 appears when you run the statement.

As shown in Figure 4-42 on page 232, this example was created using the VisualAge for Java Visual Composition Editor. We use two visual classes:

- ▶ **SQLQueryBuilderPane**: To build the SQL statements
- ▶ **SQLResultSetTablePane**: To run the statement and display the results

We use three non-visual classes: `DriverManager` from the `java.sql` package, `SQLConnection`, and `ErrorDialogAdapter`.

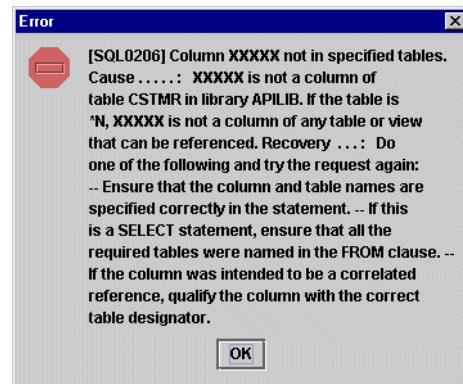


Figure 4-41 SQLQueryBuilderPane error dialog

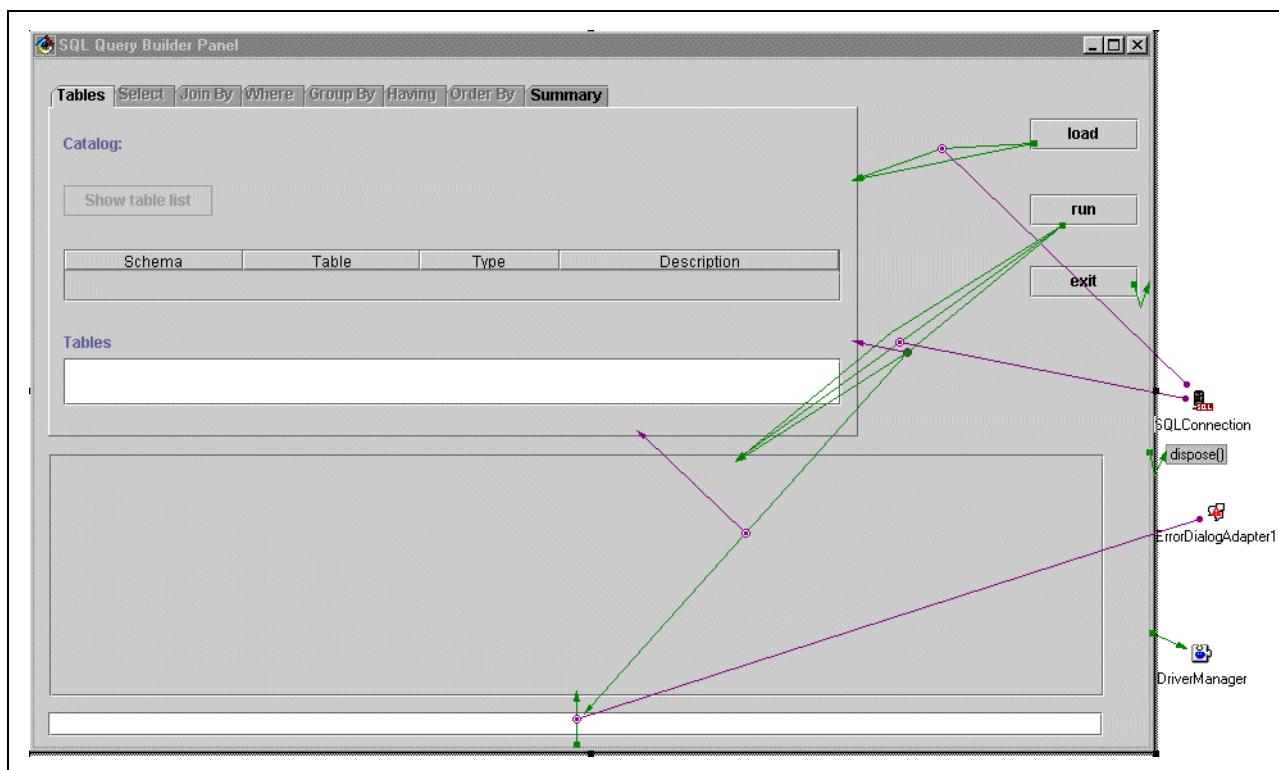


Figure 4-42 SQLQueryBuilder example in the VisualAge for Java VCE

Processing for the non-visual classes is exactly the same as shown in the SQLResultSetTablePane example discussed earlier in this chapter. Please see “SQLResultSetTablePane application” on page 226, for details.

The Load button controls the SQLQueryBuilderPane. We use the setConnection method to set the SQL connection. We pass in the SQLConnection object as the parameter for the connection to use. We use the load method to load the database schemas and tables with which we want to work. Figure 4-43 shows the Load button connections.

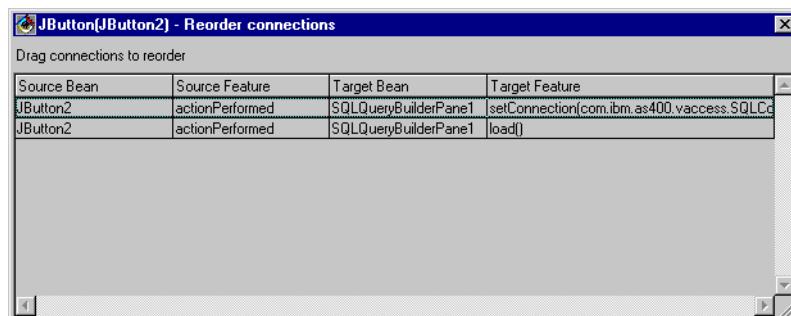


Figure 4-43 Load button connections

Figure 4-44 shows the SQLQueryBuilderPane properties dialog box. We set the tableSchema property value to APILIB. This causes all the tables in the library named APILIB to be shown.

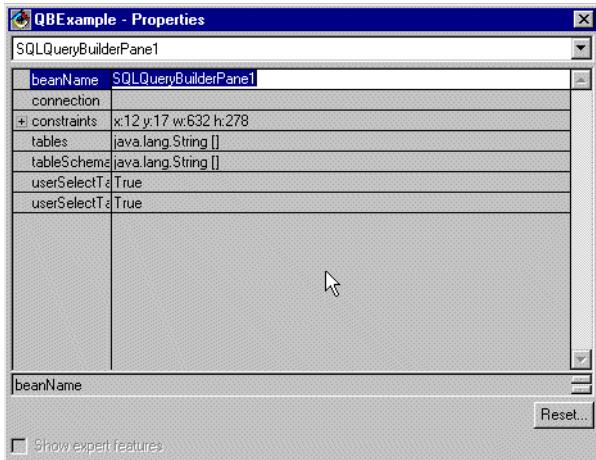


Figure 4-44 Setting the tables schema

The Run button controls the SQLResultSetTablePane. First, we set the connection property with the SQLConnect object. We use the getQuery() method of the SQLQueryBuilder pane to retrieve the query that was built and use it as the parameter for the setQuery method of the SQLResultSetTable Pane. We then use the load() method to cause the query to run and the results to be displayed. Figure 4-45 shows the Run button connections.

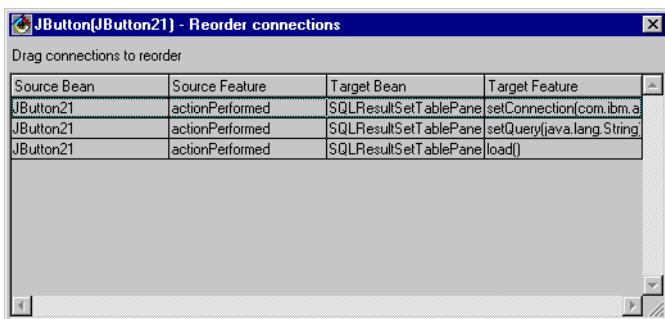


Figure 4-45 Run button connections

This completes the application. We have created an iSeries client/server application that allows us to use JDBC to interactively build and execute SQL statements against DB2 UDB for iSeries databases. We use a listener to handle and display any error conditions that occur. We did this all without writing a single line of Java code.

### 4.7.3 SQLResultSetFormPane

The SQLResultSetFormPane presents the results of a query in a form. The form provides controls that allow you to scroll forward and backward through the result set returned by the query. Figure 4-46 on page 234 shows the completed example.

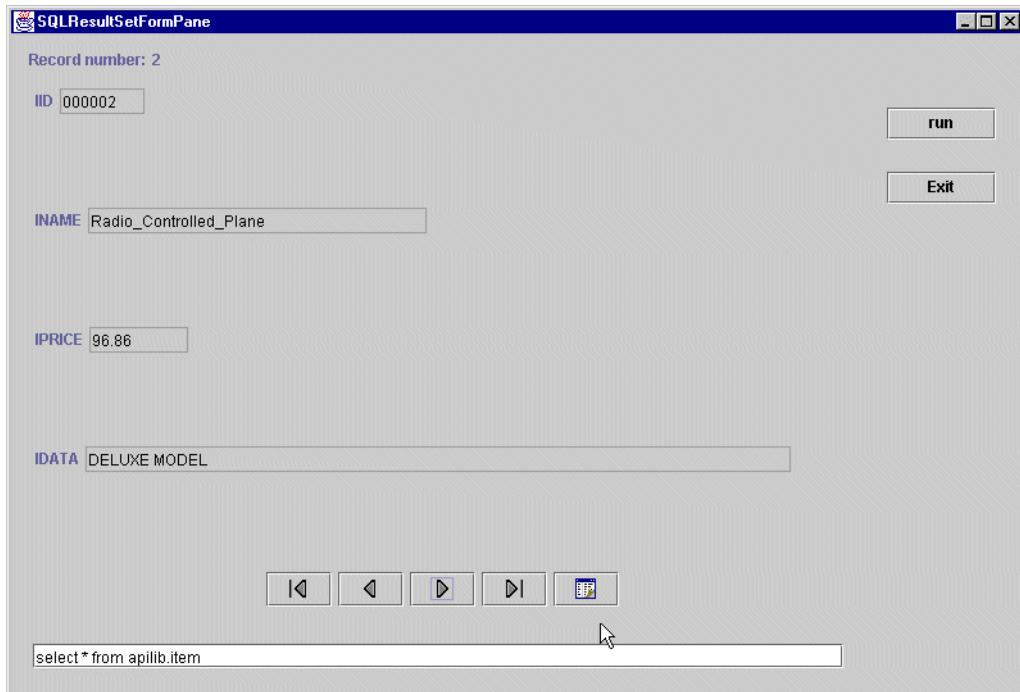


Figure 4-46 SQLResultSetFormPane example

We use a ErrorDialog Adapter to display any error conditions. If you entered an invalid column name, for example, the dialog box shown in Figure 4-47 appears.

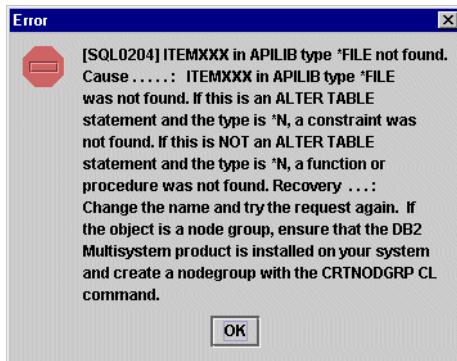


Figure 4-47 SQLResultSetFormPane example error

As shown in Figure 4-48, this example was created using the VisualAge for Java Visual Composition Editor. We use a visual component, the SQLResultSetFormPane, to execute an SQL statement and display the results. We use three non-visual classes: DriverManager from the java.sql package, SQLConnection, and ErrorDialogAdapter.

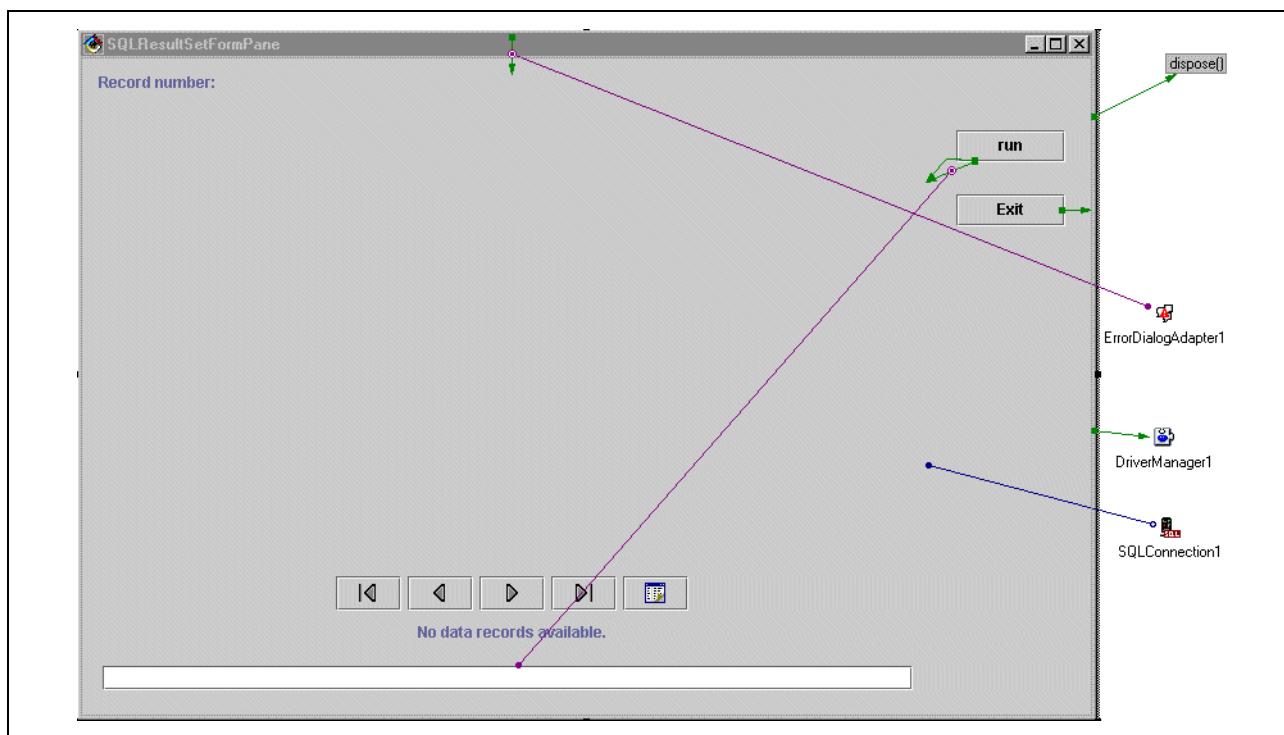


Figure 4-48 SQLResultSetForm in the VCE

Processing for the non-visual classes is exactly the same as shown in the SQLResultSetTablePane example discussed earlier in this chapter. Please see “SQLResultSetTablePane application” on page 226, for details.

The Run button controls the processing of the SQLResultSetFormPane processing. Figure 4-49 shows the Run button connections.

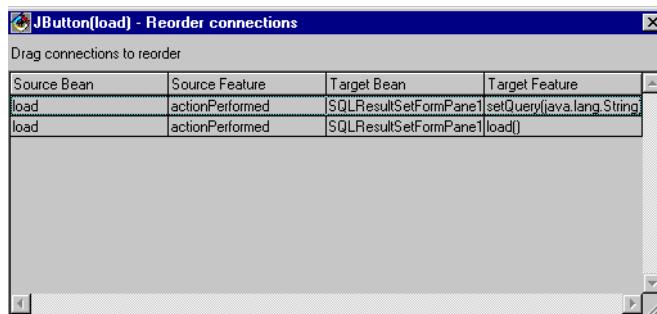


Figure 4-49 Run button for the SQLResultSetFormPane

The setQuery method is used to set the SQL statement to be executed. The value from the JTextField is used as input. The execution of the load method causes the SQL statement to be executed. The results are displayed in the SQLResultSetFormPane.

This completes the application. We created an iSeries client/server application, which allows us to use JDBC to interactively build and execute SQL statements against DB2 UDB for iSeries databases. We use a listener to handle and display any error conditions that occur. We did this all without writing a single line of Java code.

## SQLResultSetModel

The SQLResultSetModel class manages the results of an SQL query in a table. The SQLResultSetModel class allows you to have more control over the classes that you are using. In this example, we use the class in conjunction with a Swing JScrollPane. The SQLResultSetModel controls the execution and results of the SQL statement, but the JScrollPane actually displays the results. This class allows you to keep the SQL processing separate from the actual display. Figure 4-50 shows the completed example.

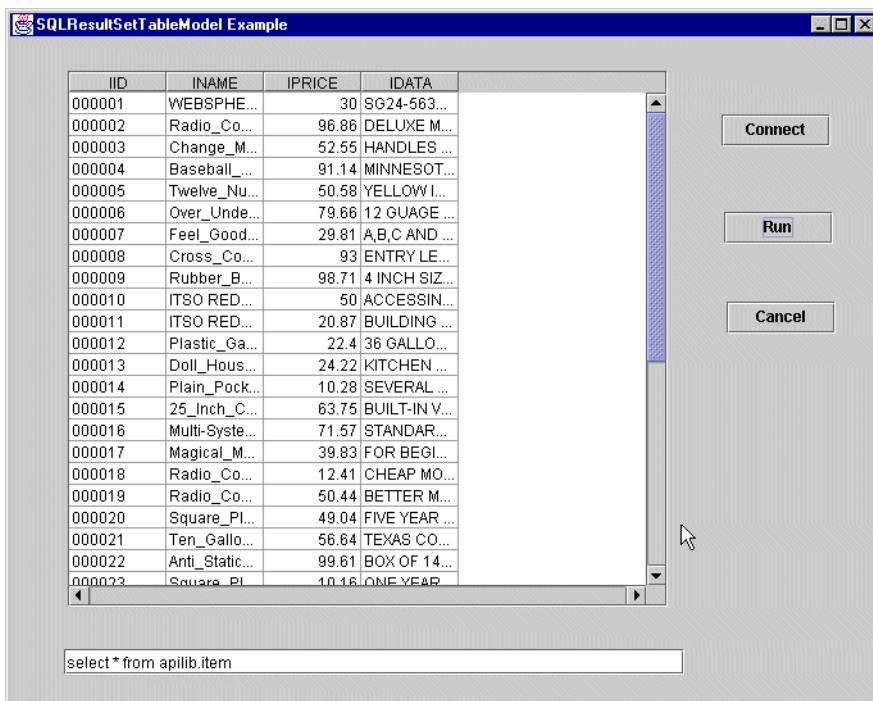


Figure 4-50 SQLResultSetTableModel example

We use an ErrorDialog adapter to display any error conditions. If you enter an invalid column name, for example, the dialog box shown in Figure 4-51 appears.

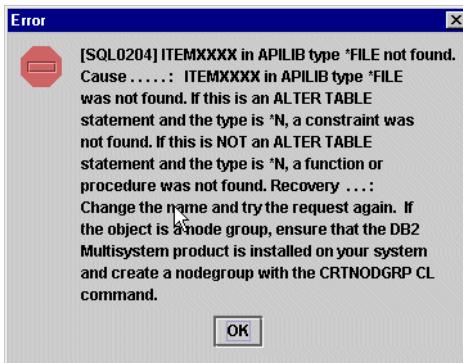


Figure 4-51 SQLResultSetTableModel error dialog

As shown in Figure 4-52, this example was created using the VisualAge for Java Visual Composition Editor. We use a visual component, the Swing JScrollPane, to display the results. We use four non-visual classes: DriverManager from the java.sql package, SQLResultSetTableModel, SQLConnection, and ErrorDialogAdapter.

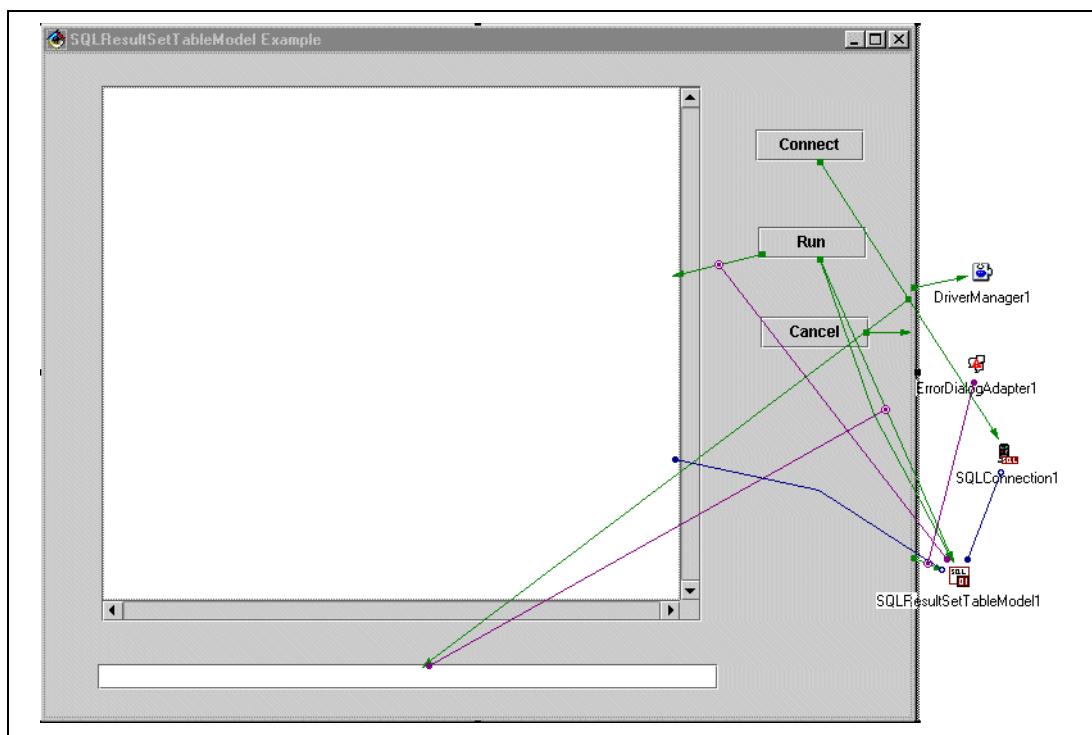


Figure 4-52 *SQLResultSetModel* in the VCE

Processing for the DriverManager, SQLConnect, and the ErrorDialogAdapter is exactly the same as shown in the *SQLResultSetTablePane* example discussed earlier in this chapter. Please see “*SQLResultSetTablePane* application” on page 226, for details.

The Run button controls the processing of the *SQLResultSetTableModel* and the *JScrollPane*. Figure 4-53 shows the Run button connections.

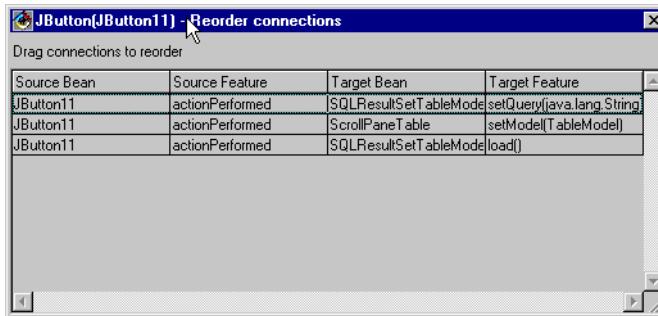


Figure 4-53 *SQLResultSetTableModel* example Run button

The *SQLResultSetTableModel* *setQuery* method is used to set the SQL statement using the *JTextField* as input. The *JScrollPane* *setModel* method is used to set the *TableModel* property to the *SQLResultSetTableModel*. The *load* method of the *SQLResultSetTableModel* is started to run the SQL statement and return the results. The results are displayed in the *JScrollPane*.

This completes the application. We created an iSeries client/server application which allows us to use JDBC to interactively build and then execute SQL statements against DB2 UDB for iSeries databases. We use a listener to handle and display any error conditions that occur. We did this all without writing a single line of Java code. This application is similar to the `SQLResultSetTablePane` example in “`SQLResultSetTablePane` application” on page 226. The difference here is that we use a Swing component, which is not part of the IBM Toolbox for Java, to display the results.

#### 4.7.4 Record-level access GUI examples

This section explains how to build iSeries client/server applications using the IBM Toolbox for Java GUI record-level access classes and VisualAge for Java. In these examples, we access the database using the DDM record-level access interface.

##### RecordListFormPane

The `RecordListFormPane` class presents a list of records from an iSeries file in a form. In this example, we use it to display the records from an iSeries file named PARTS. Figure 4-54 shows the completed example.

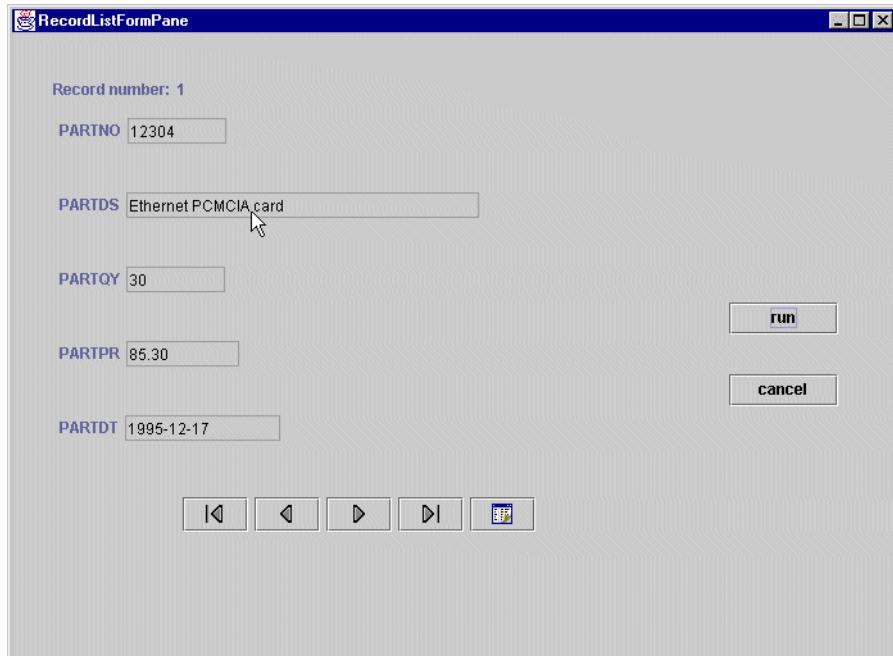


Figure 4-54 RecordListFormPane example

We use an `ErrorDialogAdapter` to view any error conditions. If you try to access a non-existent file, for example, the dialog box shown in Figure 4-55 appears.

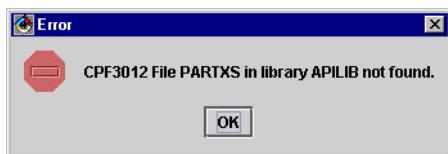


Figure 4-55 Record-level access error dialog

As shown in Figure 4-56, this example was created using the VisualAge for Java Visual Composition Editor. We use three non-visual classes: AS400, AS400FileRecordDescription, and ErrorDialogAdapter. We use a RecordListFormPane to view the records retrieved from the iSeries server.

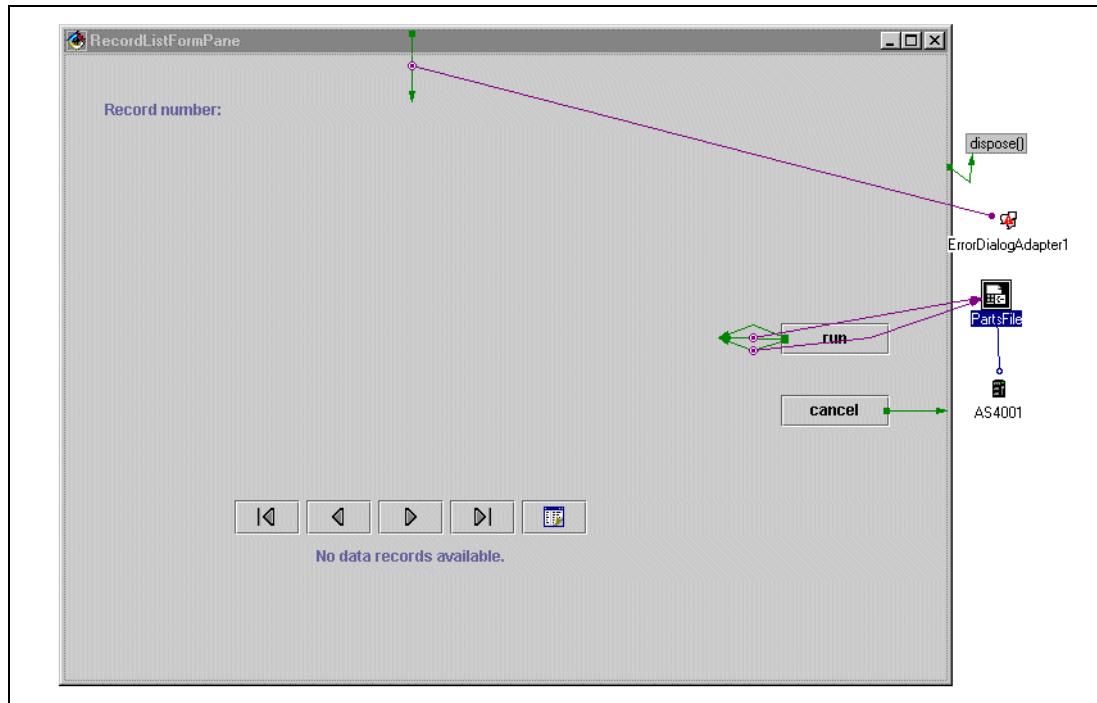


Figure 4-56 RecordListFormPane in the VCE

By dropping an AS400 object outside the frame in the VCE, we instantiate an AS400 object. We use the AS400 object for the connection to the iSeries server. We do not set the AS400 properties for system name, user ID, or password. This causes a sign-on dialog to appear the first time we try to access the iSeries server. We use the AS400FileRecordDescription object, which we name PartsFile, to control the name of the iSeries file with which we want to work.

We set the path attribute to /QSYS.LIB/APILIB/PARTS.FILE. We use iSeries IFS naming conventions to set the name of the file. The Run button controls the execution of the application. The Run button connections are shown in Figure 4-57.

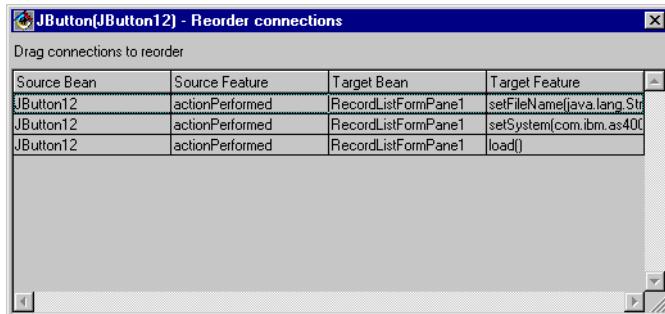


Figure 4-57 Run button connections

We use the setFileName method of the RecordListTablePane class to set the name of the file with which we want to work. We use the value from the AS400FileRecordDescription object. Next, we set the system name again using the value from the AS400FileRecordDescription object. Finally, we run the load method to cause the records to be displayed.

The final component to add is the `ErrorDialogAdapter` to handle error conditions for our application. We use the `windowOpened` event to add an `ErrorListener`. We pass in the `ErrorDialogAdapter` as a parameter to the `ErrorListener`. This is done exactly the same as in the `RecordListTablePane` example.

This completes the application. We created an iSeries client/server application that allows us to use record-level access to retrieve records from an iSeries file. We use a listener to handle and display any error conditions that occur. We did this all without writing a single line of Java code.

### RecordListFormPane using the keyed access example

In this example, we use keyed access to retrieve records by key and display them in a `RecordListFormPane`. Figure 4-58 shows the completed example.

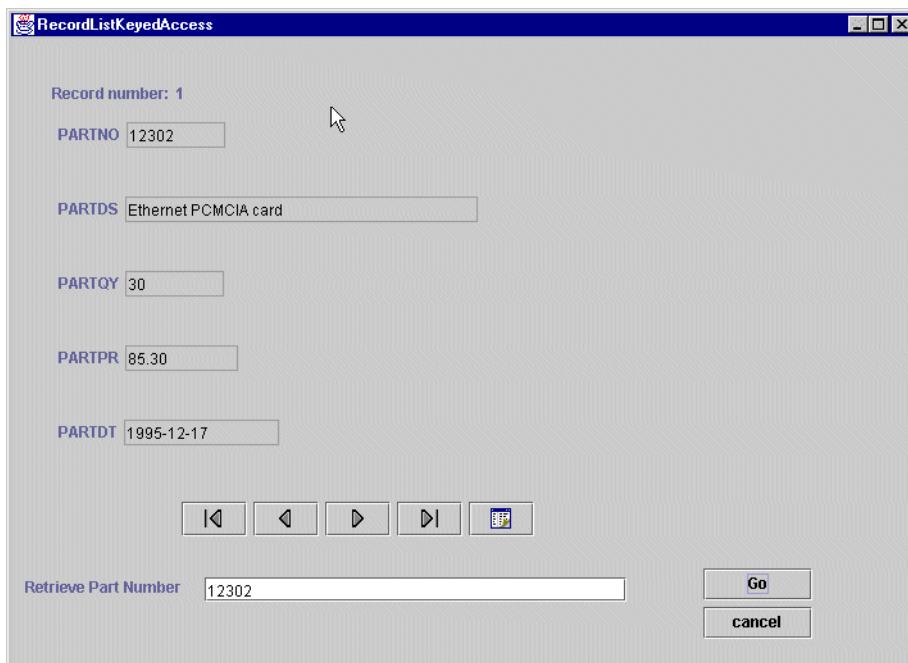


Figure 4-58 RecordListFormPane example

We use an `ErrorDialogAdapter` to display any error conditions. If you try to access a non-existent file, for example, the dialog box shown in Figure 4-59 appears.

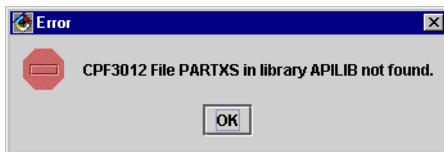


Figure 4-59 Record-level access error dialog

As shown in Figure 4-60, this example was created using the VisualAge for Java Visual Composition Editor. We use three non-visual classes: `AS400`, `AS400FileRecordDescription`, and `ErrorDialogAdapter`. We use a `RecordListFormPane` to display the records retrieved from the iSeries server.

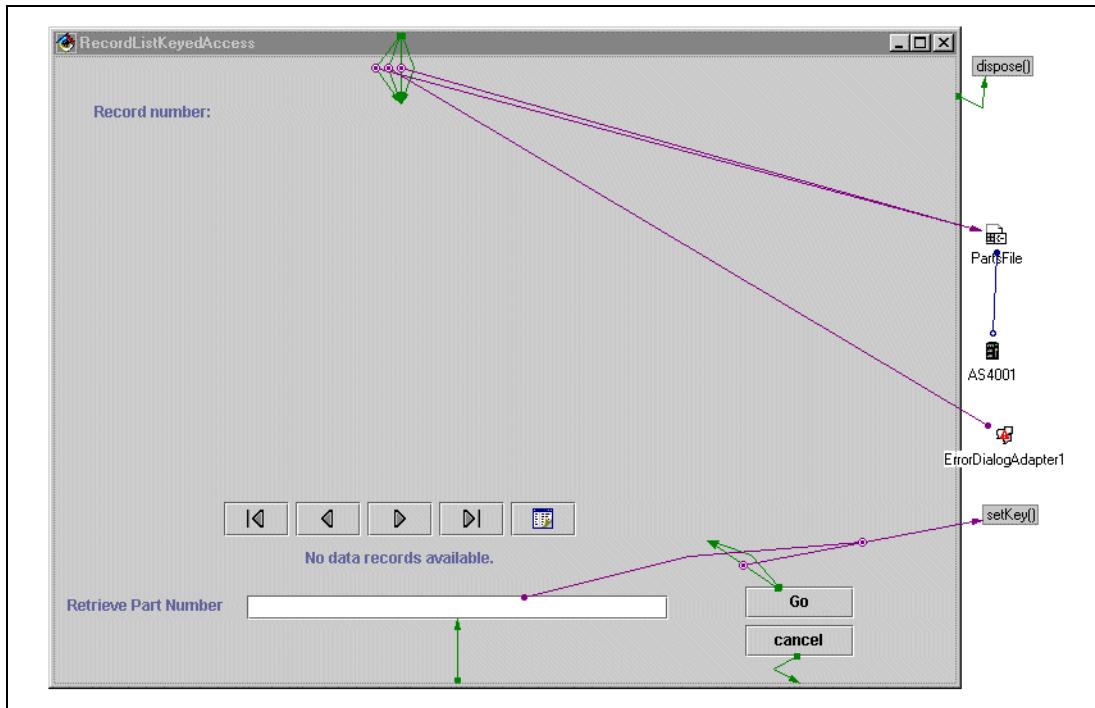


Figure 4-60 RecordListFormPane in the VCE

By dropping an AS400 object in the free-form surface of the VCE, we instantiate an AS400 object. We use the AS400 object for the connection to the iSeries server. We do not set the AS400 properties for system name, user ID, or password. This causes a sign-on dialog to appear the first time we try to access the iSeries server. We use the AS400FileRecordDescription object to control the name of the iSeries file with which we want to work.

We set the path attribute to /QSYS.LIB/APILIB/PARTS.FILE. We use iSeries IFS naming conventions to set the name of the file. This is done exactly the same as in the RecordListTablePane example.

We use the windowOpened event of the frame to control the required initialization processing. Figure 4-61 shows the window events.

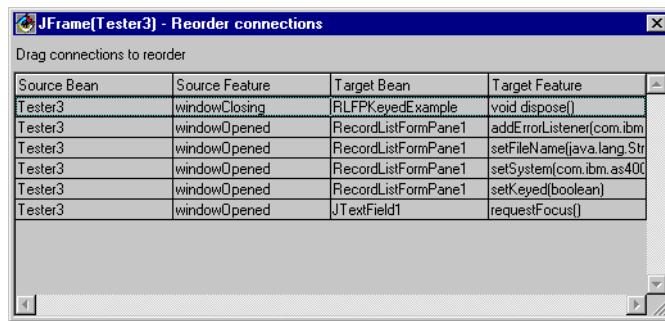


Figure 4-61 Window events connections

We add an ErrorDialogAdapter to handle error conditions for our application. We use the windowOpened event to add an ErrorListener. We pass in the ErrorDialogAdapter as a parameter to the ErrorListener.

We use the setFileName method of the RecordListTablePane class to set the name of the file with which we want to work. We use the value from the AS400FileRecordDescription object. Next, we set the system name again using the value from the AS400FileRecordDescription object. We use the setKeyed method with a value of true to specify keyed access. Finally, we request focus on the JTextField where we enter the key value of the record that we want to retrieve.

The Go button controls retrieving the record specified in the JTextField from the iSeries server. The Go button events are shown in Figure 4-62.

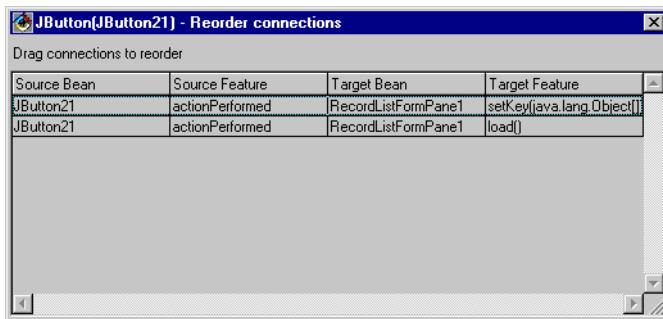


Figure 4-62 Go button connections

We provide a Java method named setKey to convert the information in the JTextField to a Object[] format, which is required by the RecordListFormPane setKey method.

```
public Object[] setKey(String partNo) {  
    Object [] theKey = new Object[1];  
    theKey[0] = new java.math.BigDecimal(partNo);  
    return theKey;.  
}
```

This completes the application. We created an iSeries client/server application that allows us to use record-level access to retrieve records from an iSeries file using keyed reads. We use a listener to handle and display any error conditions that occur.

The IBM Toolbox for Java GUI classes allow you to visually represent your iSeries data and resources. You can quickly build iSeries client/server applications using these classes. In many cases, you can do this without writing any Java code. You can also combine the GUI classes with your own application-specific code.

Java programs that use the IBM Toolbox for Java GUI classes also need Sun's JDK Swing 1.1 or later support. Swing is available with Sun's JFC. For more information about Swing, see: <http://www.javasoft.com/products/jfc/index.html>

VisualAge for Java 3.5 provides this support so you can use these classes in the VisualAge for Java Visual Composition Editor.

#### 4.7.5 Additional GUI component classes

This section describes and demonstrates some of the new or enhanced GUI classes in vaccess package of the IBM Toolbox for Java. All sample applications were written using IBM VisualAge for Java 3.5 and Modification 4 of the IBM Toolbox for Java loaded in the workspace.

## SpooledFileViewer

As the class name implies, the SpooledFileViewer class displays iSeries spooled files on a client workstation. There are two iSeries requirements for this class:

- ▶ The spooled file must reside on a V4R4 or later iSeries server.
- ▶ The AFP Utilities licensed product must be installed on the iSeries server.

The SpooledFileViewer class is a subclass of the javax.swing.JComponent. It can be added to suitable swing classes (such as JFrame). Figure 4-63 illustrates a possible use of this class. We use JSplitPane to hold an AS400DetailPane on top and a SpooledFileViewer on the bottom. The AS400DetailPane is populated with a list of iSeries spooled files. Selecting a file in the AS400DetailPane causes its contents to be displayed in the bottom pane.

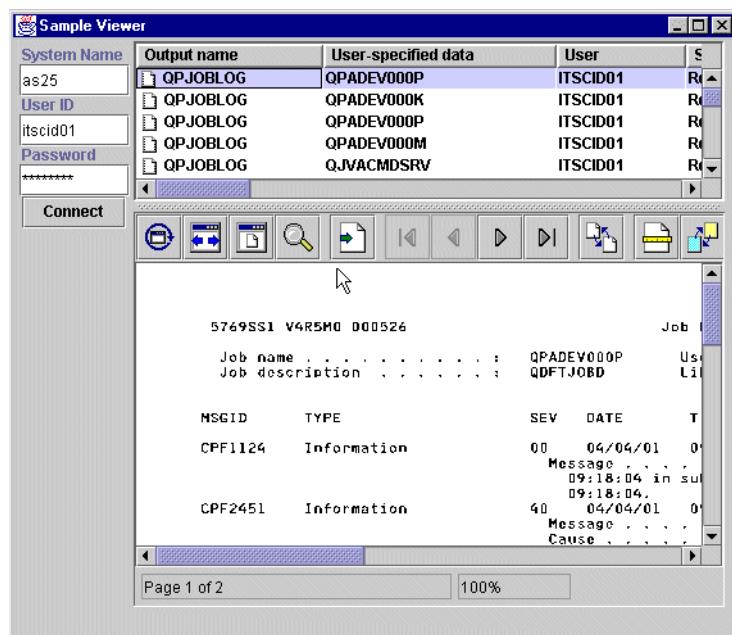


Figure 4-63 Application using the SpooledFileViewer class

### Building the spooled application

For this example, it is necessary to extend the com.ibm.as400.vaccess.SpooledFileViewer class to add two methods. We do this to enable an application to pass the AS400DetailsPlane selected object and the VPrinterObject object to the extended SpooledFileViewer class. It converts them to the required parameters for the setSpooledFile method provided with the base SpooledFileViewer class. The base SpooledFileViewer class does not directly accept AS400DetailsPanes and VPrinterObjects.

The sample application is built by extending the existing SpooledFileViewer class to provide two additional setSpooledFile methods. The extended class is used inside a JSplitPanel to create the application. The following steps show how to build the application:

1. Start IBM VisualAge for Java 3.5.
2. In TBVisual package under IBM Toolbox for Java Example project, create a new class called *ESpooledFileViewer* that extends com.ibm.as400.vaccess.SpooledFileViewer and imports the com.ibm.as400.access and com.ibm.as400.vaccess packages.
3. Create a new void method called setSpooledFile that accepts two parameters:
  - **system** of type AS400
  - **splf** of type VOutput

The method should also be capable of throwing the exception `java.beans.PropertyVetoException`.

The following code snippet forms the body of the previously created method. It uses the `VOutput` and `AS400` objects to create a `SpooledFile` object that is used to call the inherited `setSpooledFile(SpoledFile)` method.

```
String splfString = splf.toString();
String splfName = splfString.substring(0, splfString.indexOf(" "));
splfString = splfString.substring(splfString.indexOf(" ")).trim();
int splfNum = new Integer(splfString.substring(0,splfString.indexOf(" "))).intValue();
splfString = splfString.substring(splfString.indexOf(" ")).trim();
String splfUser = splfString.substring(0, splfString.indexOf(" "));
splfString = splfString.substring(splfString.indexOf(" ")).trim();
String splfJobNum = splfString.substring(0, splfString.indexOf(" "));
String splfJobName = splfString.substring(splfString.indexOf(" ")).trim();
setSpooledFile(new SpoledFile(system ,splfName, splfNum, splfJobName, splfUser,
splfJobNum));
```

4. Create another new void method called `setSpooledFile` that accepts two slightly different parameters:

- **system** of type `AS400`
- **splf** of type `VObject`

Again, the method should be capable of throwing the `java.beans.PropertyVetoException` exception.

This method simply calls the previously created method by casting the `VObject` to a `VOutput` object, as shown in the following code snippet:

```
setSpooledFile(system, (VOutput) splf)
```

Example 4-41 shows the complete `ESpooledFileViewer` class.

---

*Example 4-41 The `ESpooledFileViewer` class*

---

```
import com.ibm.as400.vaccess.*;
import com.ibm.as400.access.*;

/**
 * This type was created in VisualAge.
 */
class ESpooledFileViewer extends SpooledFileViewer {

    public void setSpooledFile(AS400 system, VObject splf) throws
        java.beans.PropertyVetoException
    {
        setSpooledFile(system, (VOutput) splf);

    }
    /**
     * Insert the method's description here.
     * Creation date: (4/8/01 6:45:45 PM)
     * @param system com.ibm.as400.access.AS400
     * @param splf com.ibm.as400.vaccess.VOutput
     * @exception java.beans.PropertyVetoException The exception description.
     */
    public void setSpooledFile(AS400 system, VOutput splf) throws
        java.beans.PropertyVetoException
    {
        String splfString = splf.toString();
```

```

String splfName = splfString.substring(0, splfString.indexOf(" "));
splfString = splfString.substring(splfString.indexOf(" ")).trim();
int splfNum = new Integer(splfString.substring(0,splfString.indexOf(" "))).intValue();
splfString = splfString.substring(splfString.indexOf(" ")).trim();
String splfUser = splfString.substring(0, splfString.indexOf(" "));
splfString = splfString.substring(splfString.indexOf(" ")).trim();
String splfJobNum = splfString.substring(0, splfString.indexOf(" "));
String splfJobName = splfString.substring(splfString.indexOf(" ")).trim();
setSpooledFile(new SpooledFile(system,splfName, splfNum, splfJobName, splfUser,
splfJobNum));
}
}

```

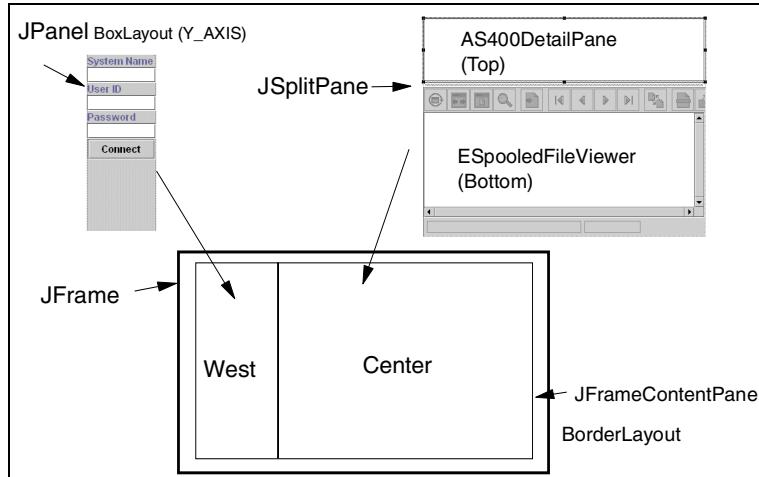
---

Next, we use the VisualAge for Java Visual Composition Editor to create an end-user application that uses the ESpoiledFileViewer class to display spooled files. Figure 4-64 shows the JavaBeans that make up the new application.

We follow this process:

1. In the VisualAge for Java IDE, create a new class named SampleViewer that extends `javax.swing.JFrame`, and select the **Compose the class visually** radio box.
2. Click **Finish** to generate the class and start the Visual Composition Editor.
3. Set the `JFrameContentPane`, inside the `JFrame`, to use the `BorderLayout`.
4. Add a new `JPanel` to the free form surface. It should not be added to the `JFrameContentPane` yet.
5. Add the Swing components (see Table 4-25) in sequence to the `JPanel`.

*Table 4-25 Swing components*



*Figure 4-64 The SampleViewer class in the IDE*

Component	Text property
JLabel	System Name
JTextField	
JLabel	User ID
JTextField	
JLabel	Password
JPasswordField	
JButton	Connect

6. Change the panel layout manager to `BoxLayout` (with the `BoxLayout axis` property set to `Y_AXIS`). It should appear the same as the panel shown in Figure 4-65 on page 246.



Figure 4-65 System and user details panel

7. Add the panel to the JFrameContentPane's west side.
8. Create a new JSplitPane in the free form surface. Again, be careful not to add it to the JFrameContentPane yet. Set the orientation property of the JSplitPane to **VERTICAL\_SPLIT**.
9. Use the Choose a bean tool to add a ESpooledFileViewer to the bottom portion of the JSplitPane object.
10. Add an AS400DetailsPane to the top portion of the JSplitPane object. Set the JSplitPane dividerLocation property to **30**. The JSplitPane should appear similar to the example in Figure 4-66.

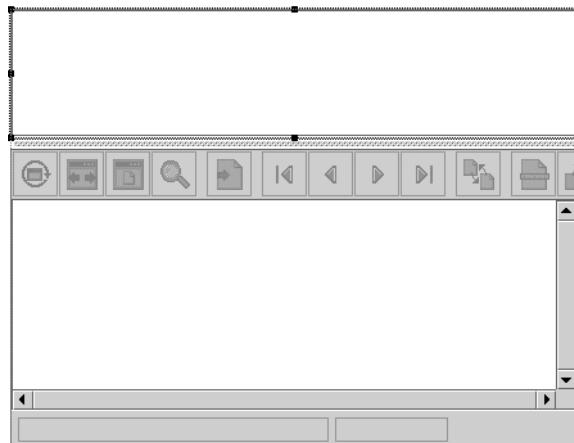


Figure 4-66 The JSplitPane

11. Add the JSplitFrame to the center of the JFrameContentPane. Again, at this time, you may need to change the JSplitPane dividerLocation property to display both panels correctly.
12. As shown in Figure 4-67, add an AS400 System and a VPrinterOutput object to the free space of the Visual Composition Editor.

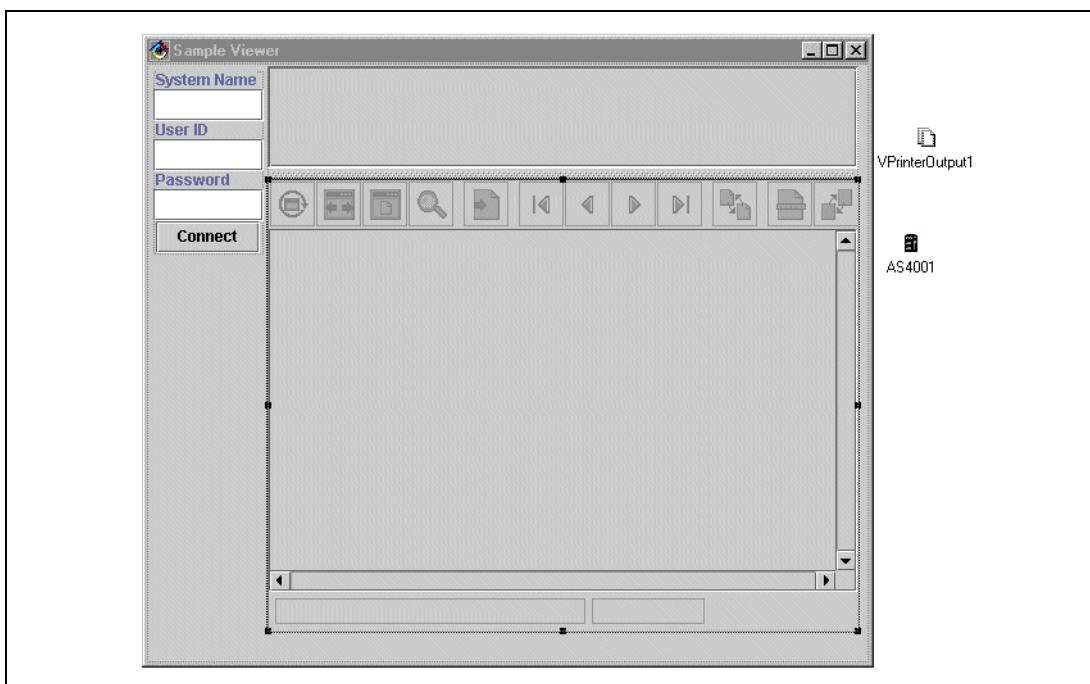


Figure 4-67 Adding AS400 and VPrinterOutput objects

13. Add the connections, in order, as shown in Table 4-26.

Table 4-26 VCE connections

From object	From event or property	To object	To property or method
JButton1	actionPerformed	AS4001	setSystemName()
Previous connection	arg1	JTextField1	text
JButton1	actionPerformed	AS4001	setUserId()
Previous connection	arg1	JTextField2	text
JButton1	actionPerformed	AS4001	setPassword
Previous connection	arg1	JPasswordField1	text
JButton1	actionPerformed	VPrinterOutput1	setSystem
Previous connection	arg1	AS4001	this
JButton1	actionPerformed	VPrinterOutput1	load()
JButton1	actionPerformed	AS400DetailsPanel1	setRoot()
JButton1	actionPerformed	AS400DetailsPanel1	load()
Previous connection	arg1	VPrinterOutput1	this
AS400DetailsPanel1	listSelection	ESpoooledFileViewer1	setSpoooledFile(AS400, VObject)
Previous connection	system	AS4001	this
Same connection as the previous connection	splf	AS400DetailsPanel1	getSelectedObject()
AS400DetailsPanel1	listSelection	ESpooledFileViewer1	load()

Figure 4-68 shows the resulting connections within the VCE.

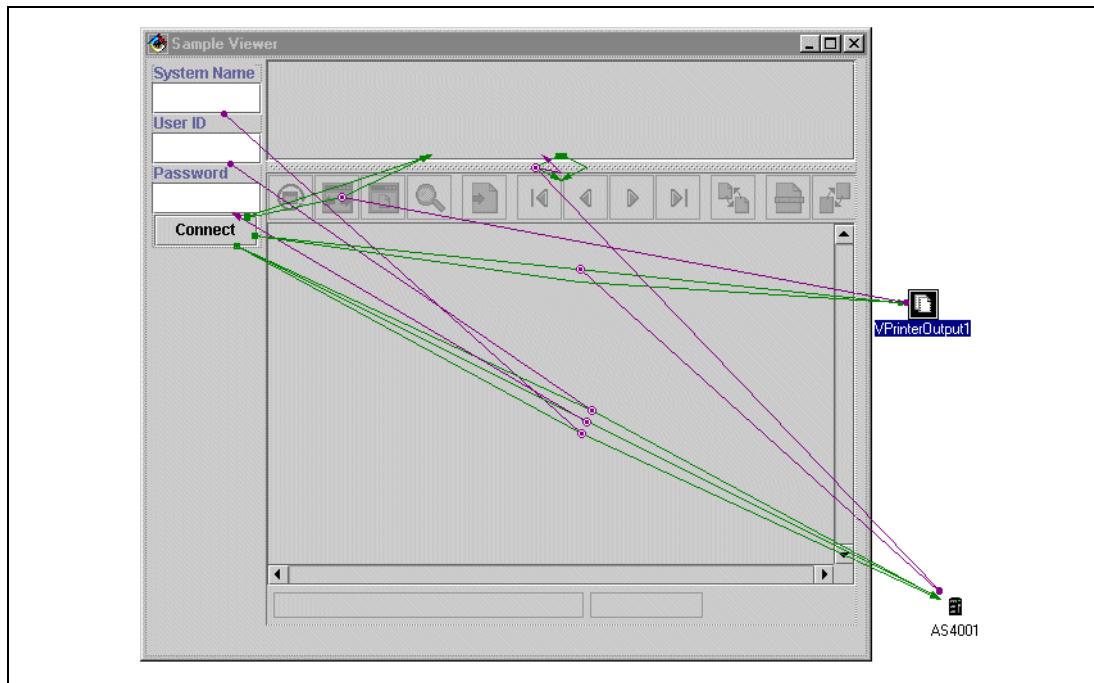


Figure 4-68 SampleViewer class with completed connections

14. Run the application.

Once it is loaded, maximize the window so you can see both subcomponents of the JSplitPane correctly.

Verify that it will connect to an iSeries server. Once the list of available spooled files is obtained from the server, selecting a spooled file in the details pane should produce the MySpooledFileViewer display as shown in Figure 4-69.

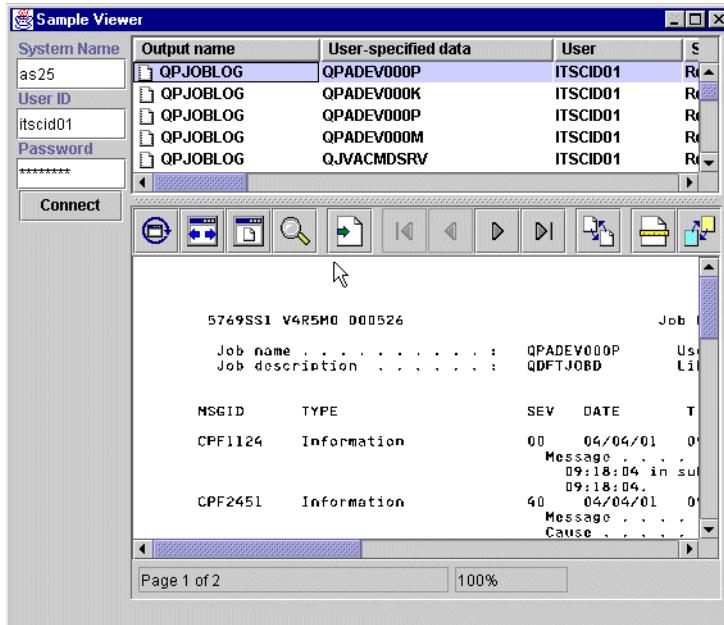


Figure 4-69 Running the application

## VSystemStatusPane

The VSystemStatusPane is another JComponent that can be added to a suitable container (such as a JFrame). As the name implies, the VSystemStatusPane is used to display information about the current system status on an iSeries server. Adding a Refresh button to the panel allows the user to monitor the iSeries status at various points in time. An example application is shown in Figure 4-70.

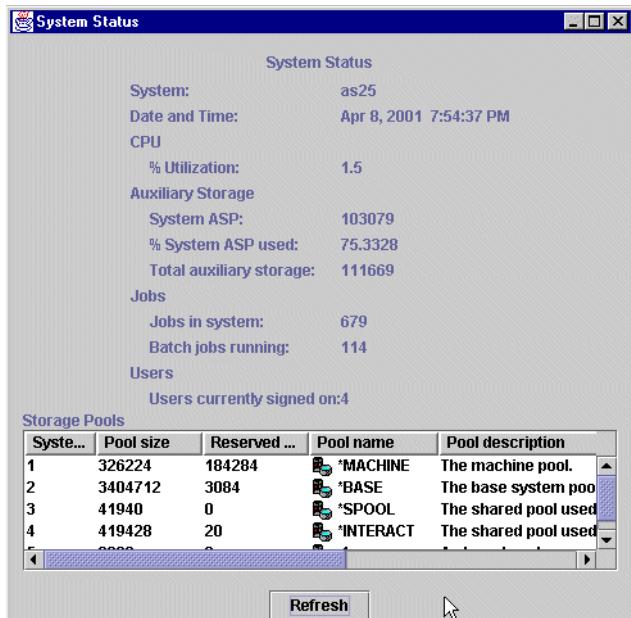


Figure 4-70 Example using the VSystemStatusPanel class

### Building the sample application

The sample application uses a bean factory to overcome a missing null constructor method for the VSystemStatusPane class. The following steps outline how to build this sample application:

1. Start VisualAge for Java 3.5.
2. Within the TBVisual package of the **IBM ToolBox for Java Example** project, create a new class called *Status*. This class should extend javax.swing.JFrame. Select the **Compose the class visually** radio box.
3. Click the **Finish** button to generate the class and start the Visual Composition Editor.
4. Set the JFrameContentPane to use the Border layout manager.
5. Add a JPanel component to the Center of the JFrameContentPane.
6. Set the JPanel to use FlowLayout.
7. Add a button to the JPanel, and set the text property to **Refresh**.
8. Set the JPanel constraints property to **South**.
9. Add an AS400 object to the free form area of the VCE.
10. Within the VCE, select the **Factory** bean (factory icon), from the **Other** palette. Add it to the free form area of the VCE.
11. Right-click the **Factory** bean and select the **Change Type...** option. Set the type to `com.ibm.as400.vaccess.VSystemStatusPane`.
12. Connect the Status initialize event to the Factory1 constructor method.

To do this, right-click the **JFrame** and select the **Connect->Connectable Features->initialize** event. Then, click the **Factory1** object, and select the **VSystemStatusPane(com.ibm.as400.access.AS400)** option.

13. Pass the AS4001 object as the parameter for the previous connection.
14. Connect the Status initialize event to the `add(Component)` method for the `JFrameContentPane`. Pass in the `Factory1` as the component parameter.
15. Connect the `actionPerformed` event of the `JButton1` object to the `load()` method for the `Factory1` object.
16. Save and run the sample application. Click the **Refresh** button to cause the sign-on dialog to appear.

## Jobs and job logs

The `VJobList` class is very simple to use. Once added to an `AS400ExplorerPane`, it can be used to display and select jobs running on an iSeries server. Selecting a job triggers the job-log messages to be displayed in the right-hand pane of the `AS400ExplorerPane`. Figure 4-71 shows the output generated once the `VJobList` loads data from an iSeries server.

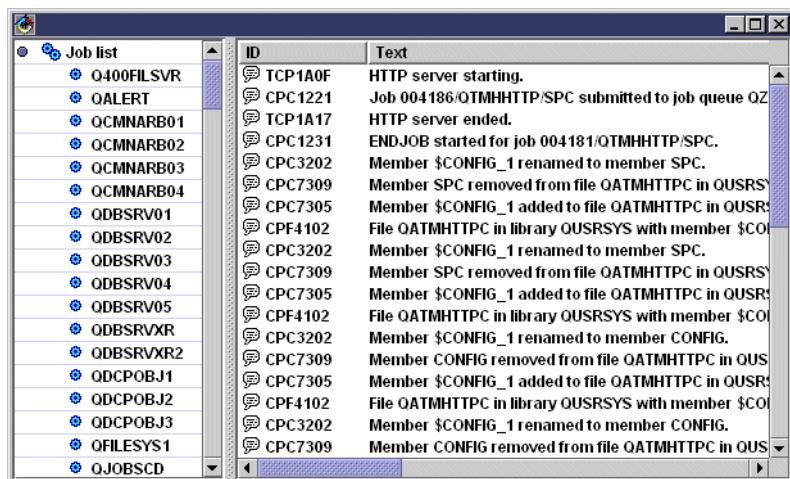


Figure 4-71 A `VJobList` example

## Building the sample

To build the `VJobList` sample, perform these steps:

1. Start IBM VisualAge for Java 3.5.
2. Within the TBVisual package of the **New Toolbox Samples** project, create a new class called *Jobs*. This class should extend `javax.swing.JFrame`.
3. Select the **Compose the class visually** radio box, and click the **Finish** button.
4. Within the VCE, set the `JFrameContentPane` to use the Border layout manager.
5. Add an `AS400ExplorerPane` to the Center of the `JFrameContentPane`.
6. Add an `AS400` object to the free form area of the VCE.
7. Add a `com.ibm.as400.vaccess.VJobList` to the free form area of the VCE.
8. Connect the job's initialize event to the `VJobList.setSystem(AS400)` method, and pass the `AS4001` object as the parameter to this method.
9. Connect the job's initialize event to the `VJobList.load()` method.

10. Connect the job's initialize event to the AS400ExplorerPane.setRoot(VNode) method, and pass the VJobList as the parameter to this method.
11. Save and run the sample application.

## Users and groups

The VUserList and VUserAndGroup classes provide a simple and efficient way to manage user profiles from a Java client. Simply adding a VUserList to an AS400DetailsPane allows the user to list iSeries user profiles.

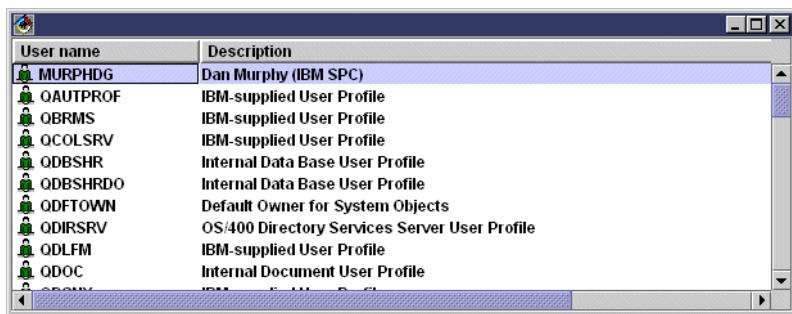


Figure 4-72 A VUserList example

To inspect user profiles in more detail, right-click the selected user profile and select the property pop-up option.

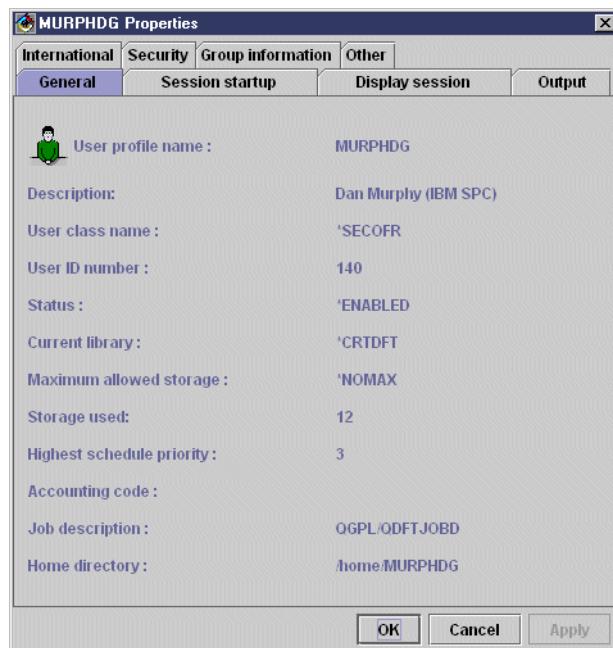


Figure 4-73 The properties for a selected user

An additional class, VUserAndGroup, allows a Java program to list user groups, all users, or users that are not in a group.

### ***Building the sample application***

The following steps create a the program used to generate the displays shown in Figure 4-72 and Figure 4-73:

1. Start IBM VisualAge for Java 3.5.
2. Within the TBVisual package of the IBM Toolbox for Java Example project, create a new class in this package called Users. This class should extend javax.swing.JFrame.
3. Select the **Compose the class visually** radio box.
4. Within the VCE, set the JFrameContentPane to use the Border layout manager.
5. Add an AS400DetailsPane to the Center of the JFrameContentPane.
6. Add an AS400 object into the free form surface.
7. Add a VUserList object to the free form surface of the VCE.
8. Connect the User object initialize event to the VUserList1 object setSystem(AS400) method.
9. Pass the AS4001 object as the parameter to the setSystem() method call generated in the previous step.
10. Connect the User object initialize event to the VUserList1 object load() method.
11. Connect the User object initialize method to the setRoot(VNode) method of the AS400ListPane1 object.
12. Set the VUserList1 object as the parameter to the setRoot(VNode) method generated in the previous step.
13. Run the program and connect to an iSeries server.

**Note:** As with all the examples in this section, the amount of time required to retrieve information from the iSeries server depends on many factors. During this time, the user is not given any indication that processing is taking place. Also, little error handling is performed. Therefore, the addition of an ErrorDialogAdaptor would improve the application's error handling. See 4.7, "Introduction to GUI component classes" on page 219, for examples of using the ErrorDialogAdapter class.

## 4.8 Introduction to proxy support

The IBM Toolbox for Java includes proxy support for some classes. Proxy support allows the processing that the IBM Toolbox for Java needs to carry out for a task to be done on a JVM that is remote to the client JVM. Proxy support includes the ability to use the SSL protocol to encrypt data.

The proxy classes reside in the jt400Proxy.jar file, which ships with the rest of the IBM Toolbox for Java. The proxy classes, like the other classes in the IBM Toolbox for Java, comprise a set of platform independent Java classes that can run on any computer with a JVM. The proxy classes dispatch all method calls to a server application, or proxy server. The full IBM Toolbox for Java classes are on the proxy server. When a client uses a proxy class, the request is transferred to the proxy server which creates and administers the real IBM Toolbox for Java objects.

Figure 4-74 shows how the standard and proxy clients connect to the iSeries server. The proxy server can be the iSeries server that contains the data.

An application that uses proxy support performs more slowly than if it uses standard IBM Toolbox for Java classes due to the extra communication needed to support the smaller proxy classes. Applications that make fewer method calls have less performance degradation.

Before proxy support, the classes containing the public interface, all the classes needed to process a request, and the application itself ran on the same JVM. When using proxy support, the public interface must be part of the application, but classes for processing requests can run on a different JVM. Proxy support does not change the public interface. The same program can run with either the proxy version of the IBM Toolbox for Java or the standard version.

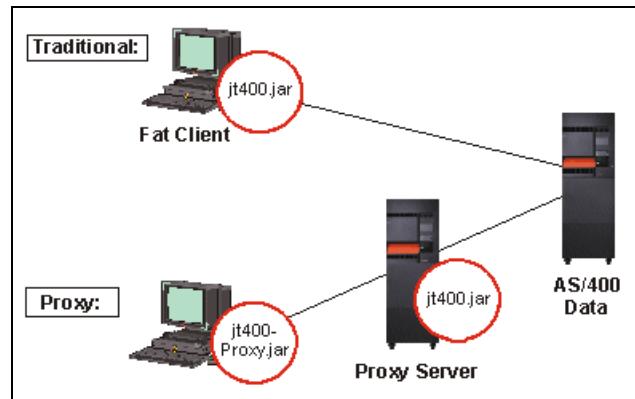


Figure 4-74 iSeries proxy support

#### 4.8.1 Classes enabled to work with proxy server

Some IBM Toolbox for Java classes are enabled to work with the proxy server application. These include:

- ▶ JDBC
- ▶ Record-level access
- ▶ Integrated file system
- ▶ Print
- ▶ Data queues
- ▶ Command call
- ▶ Program call
- ▶ Service program call
- ▶ User space
- ▶ Data area
- ▶ AS400 class
- ▶ SecureAS400 class

Other classes are not supported at this time by proxy support. Also, integrated file system permissions are not functional using only the proxy JAR file. However, you can use the JarMaker class to include these classes from the jt400.jar file.

#### 4.8.2 Proxy support example

In this section, we modify the JDBCExample example in 4.6.2, “JDBC application example” on page 153, to use proxy support. There are three machines in this example: client, proxy server, and database server. The following steps show how to set up proxy support:

1. Choose a machine to act as the proxy server. The Java environment and CLASSPATH on the proxy server machine should include the jt400.jar file. This machine must be able to connect to the iSeries server.
2. Open a Qshell session and start the proxy server on this machine by typing:

```
java com.ibm.as400.access.ProxyServer -verbose
```

Specifying *verbose* will allow you to monitor when the client connects and disconnects. See Figure 4-75 on page 254.

```

QSH Command Entry

$ 
> export -s CLASSPATH=/qibm/ProdData/Http/Public/jt400/lib/jt400.jar:$CLASSPATH
$ 
> java com.ibm.as400.access.ProxyServer -verbose
Proxy server started.
Proxy server listening to port 3470.

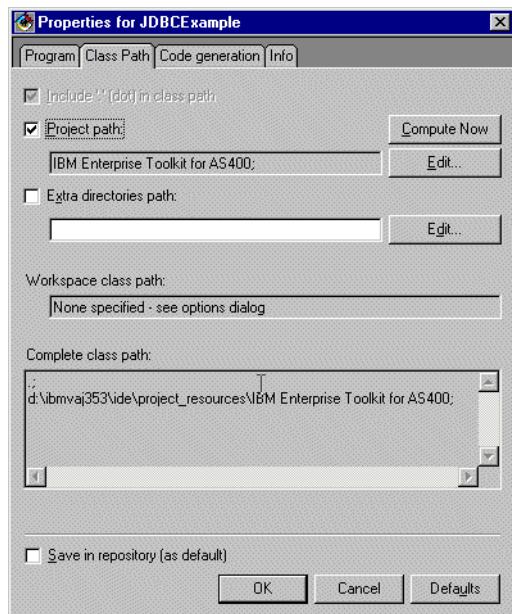
====>

F3=Exit F6=Print F9=Retrieve F12=Disconnect
F13=Clear F17=Top F18=Bottom F21=CL command entry

```

*Figure 4-75 Starting the proxy server*

3. On the client machine, the Java environment and CLASSPATH should include the jt400Proxy.jar file and your application classes. In the VisualAge for Java 3.5 IDE, you can specify the IBM Enterprise Toolkit for AS400 package in your Class Path. See Figure 4-76. This machine must be able to connect to the proxy server but does not need a connection to the iSeries server.



*Figure 4-76 Setting the classpath for proxy support*

4. Set the value of the com.ibm.access.AS400.proxyServer system property to be the name of your proxy server (refer to Figure 4-77), and run the application.

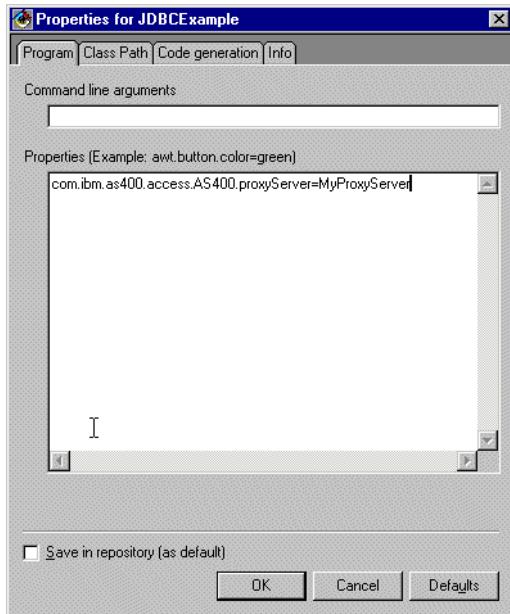


Figure 4-77 Setting the properties for proxy support

- When you click the **Connect** button, you should see (if you specified verbose mode in step 2) the application make at least one connection to the proxy server. See Figure 4-78.

```
QSH Command Entry

$ 
> export -s CLASSPATH=/qibm/ProdData/Http/Public/jt400/lib/jt400.jar:$CLASSPATH
$ 
> java com.ibm.access.ProxyServer -verbose
Proxy server started.
Proxy server listening to port 3470.
Proxy server accepted connection requested by p23m2548.rchland.ibm.com/9.5.62
.33 as connection 1001.

====>

F3=Exit F6=Print F9=Retrieve F12=Disconnect
F13=Clear F17=Top F18=Bottom F21=CL command entry
```

Figure 4-78 Connecting to the proxy server

- After the connection is established successfully, you can run your application's functions, such as Get Part. See Figure 4-79 on page 256.

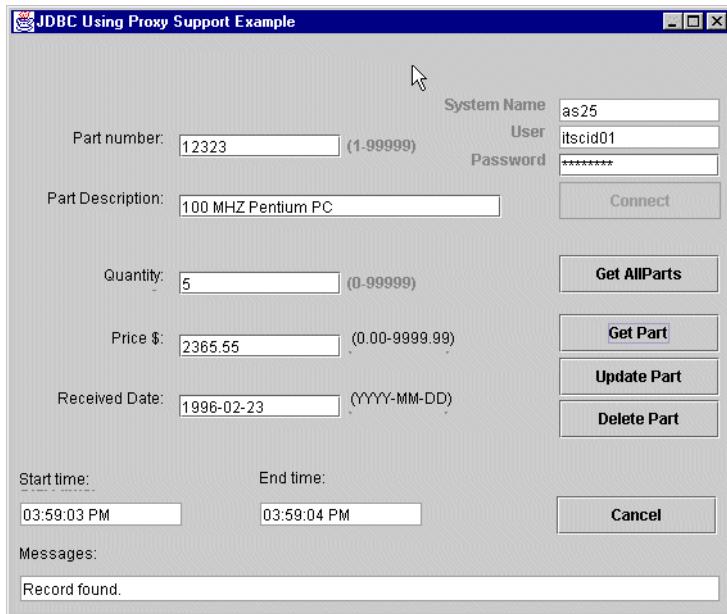


Figure 4-79 JDBC using proxy support example

## 4.9 Conclusion

The IBM Toolbox for Java is a library of Java classes supporting the client/server and Internet programming models to an AS/400 or iSeries server. The classes can be used by Java applets, servlets, and applications to access data and resources on the server. The Toolbox does not require additional client support over and above what is provided by the JVM and TCP/IP. JTOpen is the open source version of the product. It runs on a wide variety of platforms, including AIX, AS/400 and iSeries servers, Linux, Network Station, OS/2, Solaris, and Windows.



# IBM Enterprise Toolkit for AS/400

IBM Enterprise Toolkit for AS/400 (ET/400) is included in both the VisualAge for Java Enterprise Edition and VisualAge for Java for iSeries. ET/400 provides iSeries-unique Java support. It provides SmartGuides that allow you to use IBM Toolbox classes easier (Program Call SmartGuide). Plus, SmartGuides make it easy to develop and implement iSeries Java programs using VisualAge for Java. The ET/400 support is integrated into the VisualAge for Java 3.5 Integrated Development Environment. ET/400 includes:

- ▶ The IBM Toolbox for Java classes that can be used to access iSeries resources and services. These are the same classes shipped with the IBM Toolbox for Java product in iSeries server
- ▶ A SmartGuide to generate Java classes and beans for remote program calls to iSeries programs written in ILE RPG, ILE C, or OPM COBOL
- ▶ A SmartGuide to convert iSeries display file records to Java Swing classes
- ▶ iSeries beans, including JFormat beans, Data File Utility (DFU) beans, and Object List beans to access data and objects in iSeries server
- ▶ Support to export, compile, run iSeries Java applications from the VisualAge for Java Integrated Development Environment (IDE)

## 5.1 Using ET/400

The ET/400 tools are available from the VisualAge for Java Integrated Development Environment. To access the ET/400 tools, select **ET/400** from the **Tools** pop-up menu option as shown in Figure 5-1.

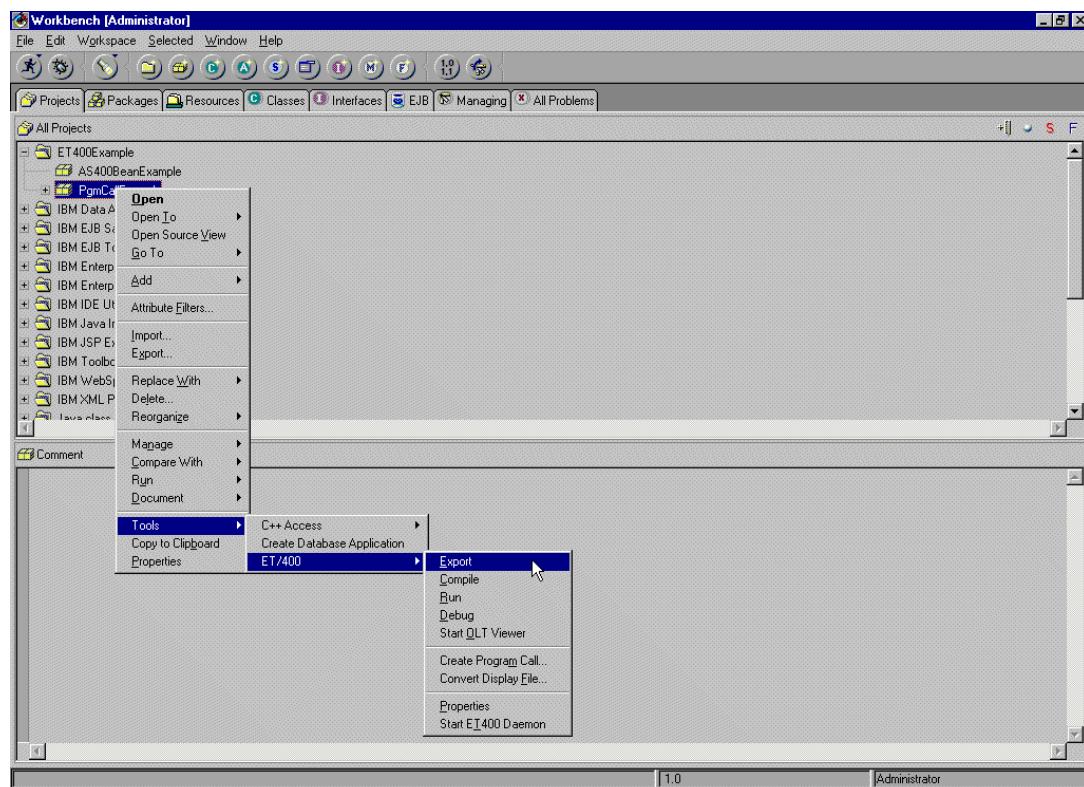


Figure 5-1 Accessing ET/400

## 5.2 IBM Toolbox for Java classes

If you want to use the IBM Toolbox for Java classes inside the VisualAge for Java Integrated Development Environment, you must import the classes inside the IDE. ET/400 simplifies this process. After you install VisualAge for Java 3.5 Enterprise Edition or VisualAge for Java for iSeries, the Toolbox classes are available in the repository as part of the IBM Enterprise Toolkit for AS400 project. If you want to use the Toolbox classes, perform these steps:

1. From the workbench, click **File-> Quick Start**.
2. Click **Features-> Add Feature**, and then click **OK**.
3. Select **IBM Enterprise Toolkit for AS400 V3.5**, and click **OK**.

This adds the Toolbox classes to your workspace. The IBM Enterprise Toolkit for AS/400 is listed under All Projects.

The alternative is to perform these tasks:

1. Install the IBM Toolbox for Java on an iSeries server.
2. Download the classes (jt400.jar or jt400.zip) to your workstation.
3. Import the classes into the VisualAge for Java IDE.

## 5.3 Distributed Program Call SmartGuide

The IBM Toolbox for Java Distributed Program Call feature allows you to call iSeries programs from a client Java application. The Create Program Call SmartGuide simplifies using this interface. With the Create Program Call SmartGuide, it is not necessary to code the program calls. You type in the iSeries server name, the name of the iSeries server program your Java program will call, and the parameters you want to pass. The SmartGuide generates the Java code for you. The data conversions between the iSeries data types and the Java data types are handled for you through the IBM Toolbox for Java classes.

### 5.3.1 Distributed Program Call feature

The Distributed Program Call feature of the IBM Toolbox allows a Java program to directly run any non-interactive program object (\*PGM) on the iSeries server. It passes input data as parameters and returns results through parameters.

The Java developer must use the data conversion classes from the Toolbox to convert input parameters from a Java format to an iSeries data type and convert output parameters from an iSeries format to a Java format.

The advantage of using the Distributed Program Call class is that native iSeries non-interactive programs can be run from a Java application unchanged. Native program calls can also result in better performance of a Java application when compared with JDBC. Additionally, this interface can call programs on the iSeries server that do more than just database access. For example, a Java application can call a program that starts nightly job processing, saves libraries to tape, or sends or receives data through communication lines.

To call a native iSeries program, you must follow these steps:

1. Connect to the iSeries server by creating an AS400 object.
2. Create a ProgramCall object.
3. Define and initialize a ProgramParameter array for passing parameters to or from the called program.
4. Use the Data Conversion classes to convert input parameter values from a Java format to an iSeries format.
5. Use the setProgram method to specify the qualified name of the program to call and parameters to use, if they are not declared on the ProgramCall constructor.
6. Execute the program using the run method.
7. If the run method fails, obtain detailed error information through AS400Message objects.
8. Retrieve output parameters using the getOutputData method of the ProgramParameter object.
9. Convert output parameter values using the data conversion classes.

### 5.3.2 Application description

This example uses the Distributed Program Call (DPC) interface (see Figure 5-2 on page 260) to allow a client program to call an iSeries program.

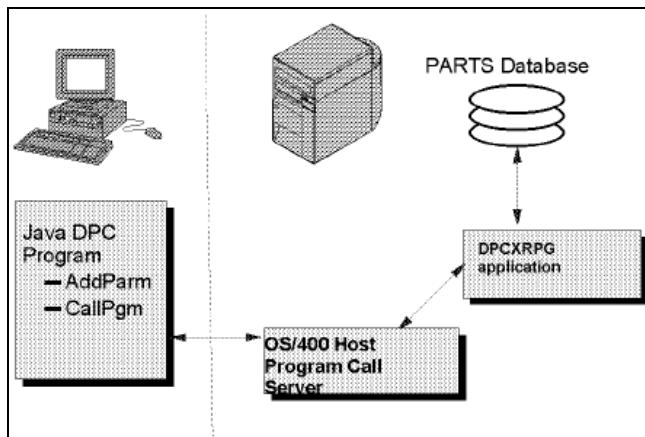


Figure 5-2 Distributed Program Call example

The client program requests data from the iSeries database by calling an iSeries program. Information is passed between the programs using parameters. It is up to the application implementer to handle data conversions.

The client program requests data from the server program by calling it and passing it parameters. The input parameters are a flag, a part number, and the attributes of a part. For example, S12323 is a request for a single record(Flag = S) of part number

12323. If requesting all parts (Flag=A), the part number is not necessary. The server program, DPCXRPG, searches the database for the requested information. The result is passed back in the output parameters. See Figure 5-3.

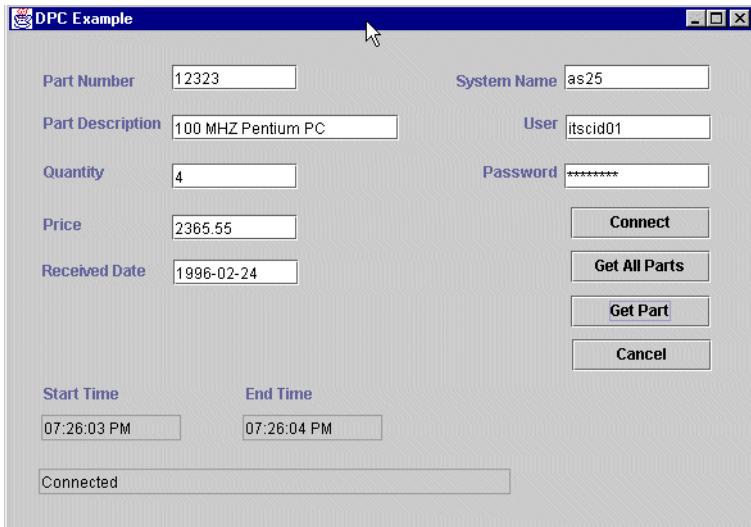


Figure 5-3 Distributed Program Call example

## RPG program background

Library: APILIB

Program Name: DPCXRPG

Parameters (all are used as Input/Output):

Table 5-1 Parameter list

Sequence/field	Description	Length/type
1 / OPTION	In: Operation Code / Out: Return Code	1 / character
2 / PARTNO	Part Number	5.0 / packed
3 / PARTDS	Part Description	25 / character
4 / PARTQY	Part Quantity	5.0 / packed

Sequence/field	Description	Length/type
5 / PARTPR	Part Price	6.2 / packed
6 / PARTDT	Part Received Date	10 / date

Values of the Operation Code(Input OPTION):

*Table 5-2 Flag operation codes*

Operation code	Database operation to execute
S	Retrieve a single record for the supplied key
A	Retrieve all records
F	Fetch next record based on the current position
E	End the program
D	Delete a single record for the supplied key
U	Update a single record for the supplied key with the attribute data. Write a single record for the supplied key with the attribute data if it doesn't yet exist.

Values of the Return Code(Output OPTION):

*Table 5-3 Flag operation codes*

Return code	Result description
Y	Normal: Operation has succeeded / When operation code was U: Record Updated
X	Normal: Operation has failed / When operation code was U: Record Added
U	Unknown operation code has been supplied

### 5.3.3 Creating a Program Call JavaBean

As shown in Figure 5-4 on page 262, to start the ET/400 Program Call SmartGuide, we highlight a package and right-click to select Create Program Call. The output is a new class or JavaBean in the selected package.

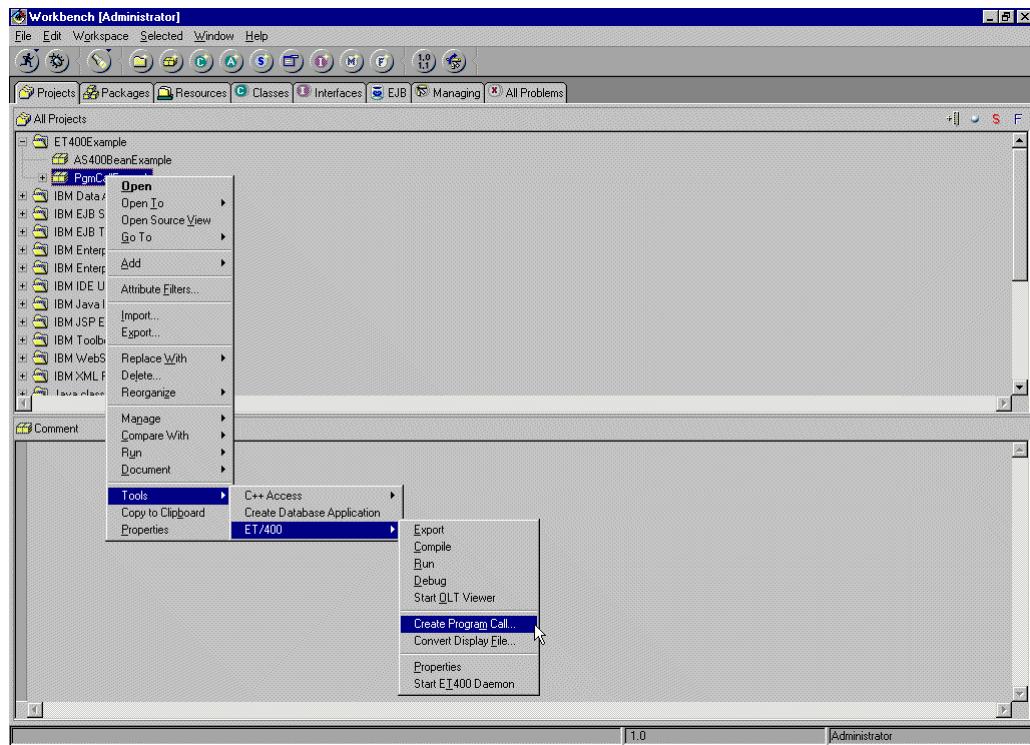


Figure 5-4 ET/400 Create Program Call

As shown in Figure 5-5, the first iSeries Program Call SmartGuide window allows us to choose whether we want to create a new class or modify an existing class. We choose **Create a new program call class**, and click **Next**.

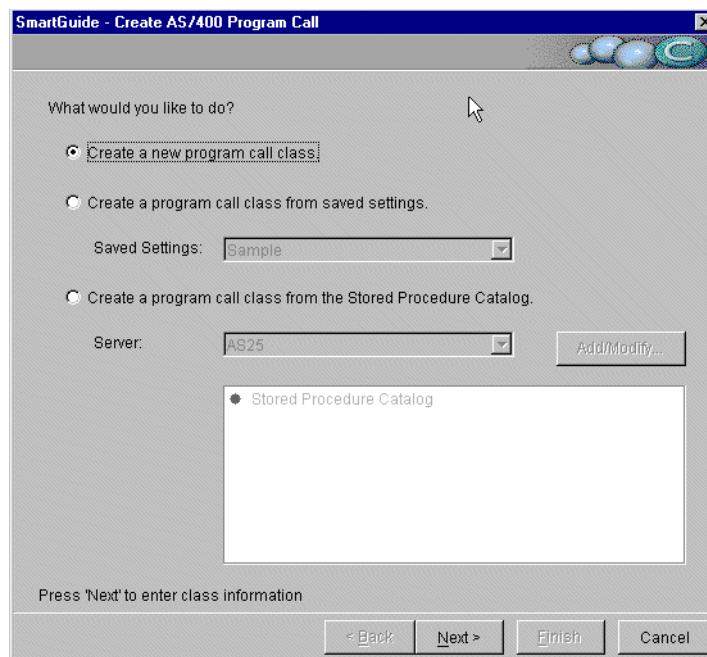


Figure 5-5 SmartGuide – Create iSeries Program Call (Part 1 of 3)

Next, we specify the name of the iSeries server, the name of the program, and the library where the program is found. We can choose the Browse button, and the SmartGuide interactively retrieves the programs from the specified library. We then select the program that we want to call. Once we enter the program name to call, we enter the name of the JavaBean or class that we want to generate and the project and package in which to store it. See Figure 5-6.

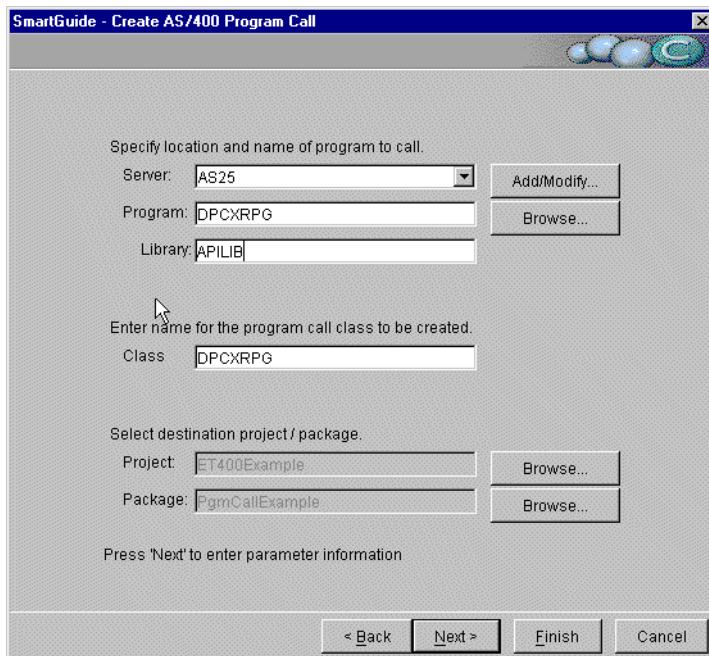


Figure 5-6 SmartGuide – Create iSeries Program Call (Part 2 of 3)

Finally, we need to define the parameters that the iSeries program uses. The SmartGuide provides choice boxes to insure that we configure the parameters properly. Since this is an iSeries RPG program, the SmartGuide does not have any way of knowing what the required parameters are. We need to check the iSeries program to determine this. In Figure 5-7 on page 264, we show the parameters required for the iSeries RPG Program named DPCXRPG.

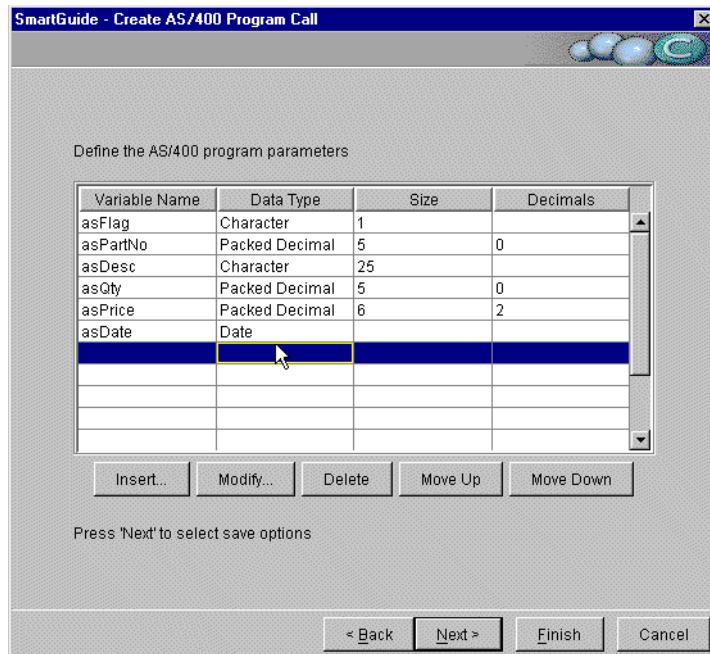
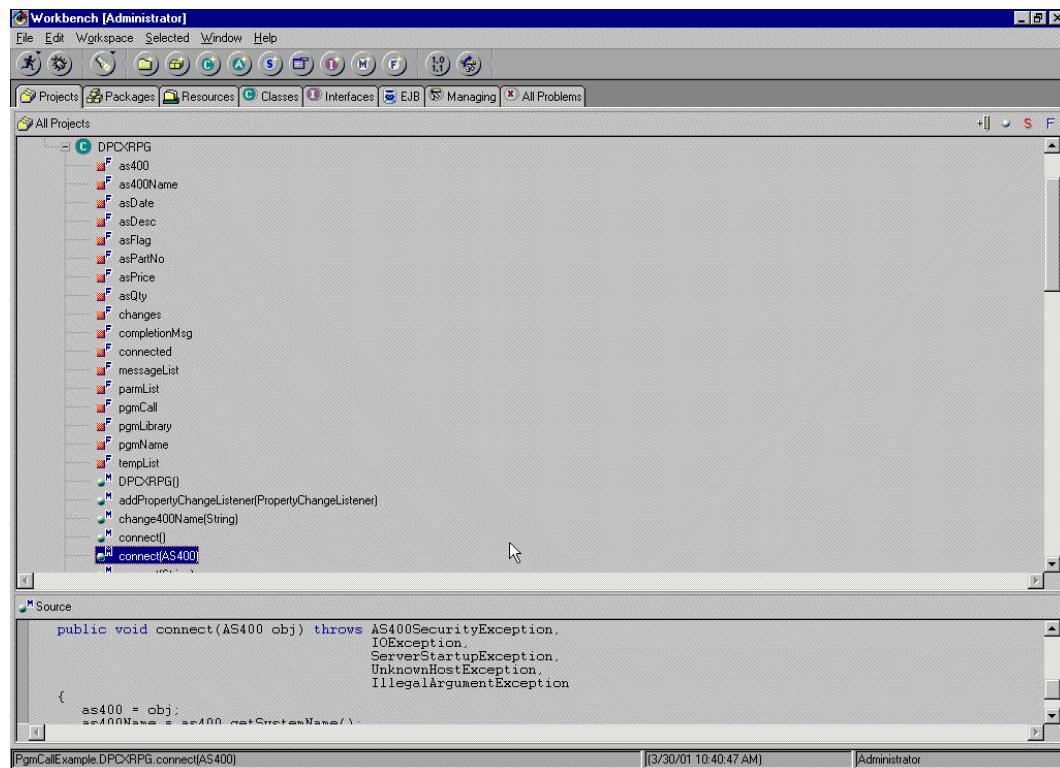


Figure 5-7 SmartGuide – Create iSeries Program Call (Part 3 of 3)

Click the **Finish** button to generate the new bean. In this case, a bean named DPCXRPG is generated. A number of fields and methods are generated that allow us to effectively use the generated bean. The generated fields and methods are shown in Figure 5-8.



*Figure 5-8 DPCXRPG generated methods*

The constructor method generated, DPCXRPG(), does not initialize the date properly. This causes a problem for the host iSeries RPG program. We fix this problem by changing the line of code in the DPCXRPG() constructor method from:

```
setasDateAsString (" ");  
to:  
SetasDateAsString("2001-01-01");
```

The SmartGuide generates several connection methods for connecting to the iSeries server. It does not generate a connect method that takes three parameters (system name, user ID, and password). We add such a method to the DPCXRPG bean to make our application development easier. The new connect method is shown in Example 5-1.

*Example 5-1 New connect source code*

```
public void connect(String name, String user, String password) throws java.lang.Exception  
{  
    as400 = new AS400(name,user,password);  
    as400Name = name;  
    connect();  
}
```

This user-supplied connect method accepts system name, user ID, and password as parameters. We can use this method in the Visual Composition Editor and visually pass in these parameters from screen text fields.

### 5.3.4 Building an application using the DPCXRPG bean in the VCE

You can use the DPCXRPG bean as a non-visual part in a visual editor. This section describes how to use the generated bean as a non-visual component in the VisualAge for Java Visual Composition Editor. We create a class named DPCEExample. We open the Visual Composition Editor and build the graphical user interface. We add a DPCXRPG bean as a non-visual part, called aDPCXRPG. This has the effect of instantiating a DPCXRPG object, which we can use in the visual builder. This is shown in Figure 5-9.

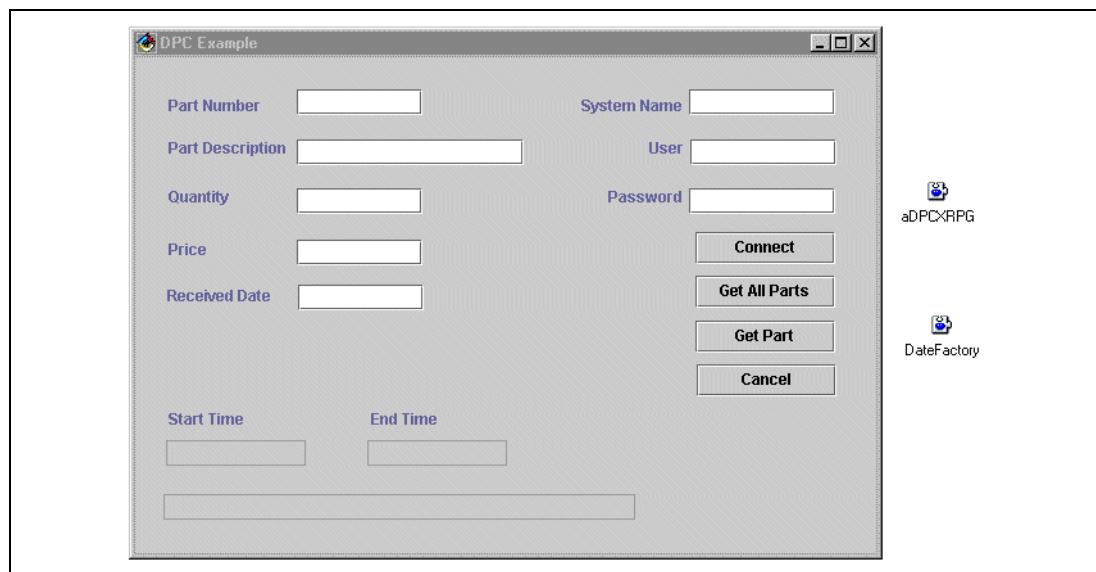


Figure 5-9 Building the graphical user interface

We can use the VCE to display the methods available with DPCXRPG. To handle the event generated when a user clicks the Connect button, we connect the actionPerformed event of the button to the aDPCXRPG bean's connect method which uses three parameters. The aDPCXRPG bean's methods are shown in Figure 5-10.

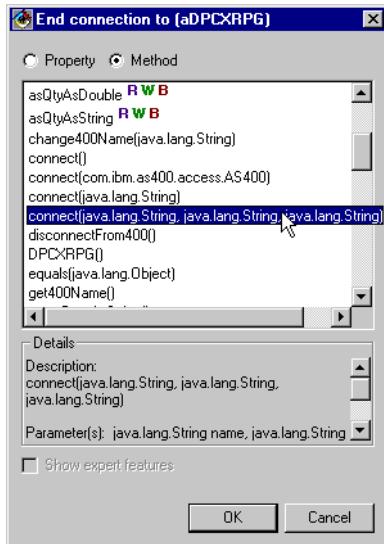


Figure 5-10 DPCXRPG methods

As shown in Figure 5-11, we pass in the three parameters. We connect the System Name, User, and Password JTextFields fields text property to the connecting line between the connect button and the aDPCXRPG bean.

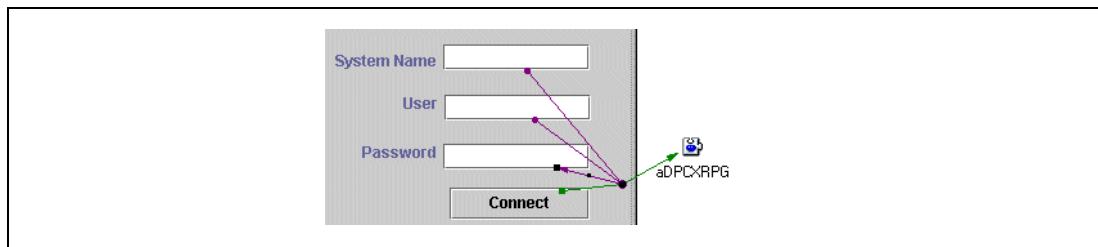


Figure 5-11 Passing in the parameter to the connect method

Figure 5-12 shows the Cancel button connections. We connect the ActionPerformed event of the Cancel button to the aDPCXRPG bean's disconnectFromAS400 method. We also connect the ActionPerformed event of the Cancel button to the dispose method of the frame.

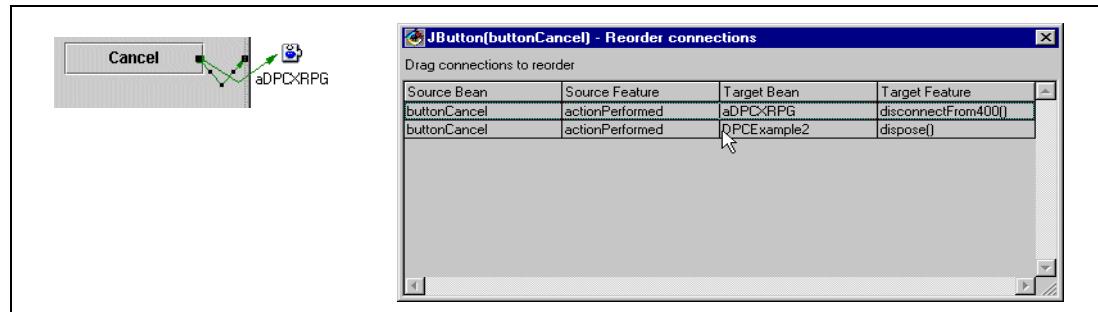


Figure 5-12 Cancel button processing

To handle the action performed when the user clicks the Get Part button, we connect the ActionPerformed event of the Get Part button to the DPCXRPG bean's `setasFlagAsString` method and supply the connection with the parameter value "S". To set the parameter, we double-click the connection line and click Set parameters. Next, we connect the ActionPerformed event of the button to the DPCXRPG bean's `setasPartNoAsString` method and supply the connection with the parameter value of the text field Part number. Finally, we connect the ActionPerformed event of the button to the bean's `runProgram` method. These connections are shown in Figure 5-13.

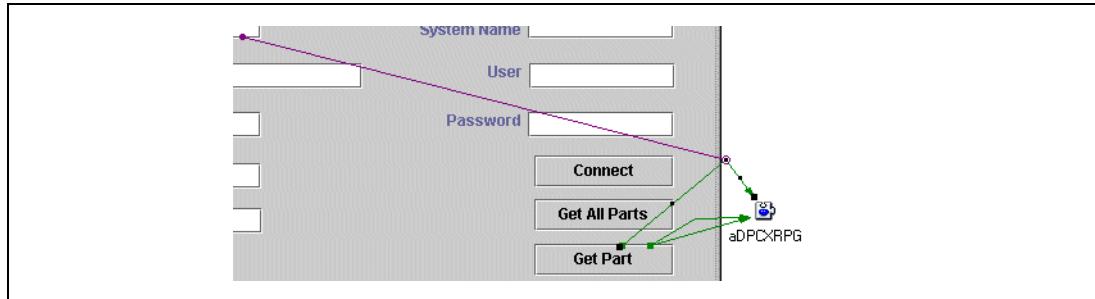


Figure 5-13 Get Part Button processing

To set the other parameters, perform the following steps:

1. Move the mouse pointer on top of the last connection line (for `runProgram`).
2. Right-click and select **Connect** and **NormalResult** from the pop-up menu that appears.
3. Drag the mouse and click the **Part Description** text field. Select **Text** from the pop-up menu.
4. Move the mouse pointer on top of the new connection line.
5. Right-click and select **Connect** and **Value** from the pop-up menu that appears.
6. Drag and click the **aDPCXRPG** bean. Select **All Features** from the pop-up menu and the `getasDescAsString()` method.
7. Click **OK**.

Repeat steps one through seven for the rest of the text fields in the part using the appropriate getter methods from the **aDPCXRPG** bean. The completed Get Part button's connections appear as shown in Figure 5-14.

Source Bean	Source Feature	Target Bean	Target Feature
buttonGetPart	actionPerformed	DateFactory	Date()
buttonGetPart	actionPerformed	aDPCXRPG	setasFlagAsString(java.lang.String)
buttonGetPart	actionPerformed	aDPCXRPG	setasPartNoAsString(java.lang.String)
buttonGetPart	actionPerformed	aDPCXRPG	runProgram()
buttonGetPart	actionPerformed	DateFactory	Date()

Figure 5-14 Get Part connections

In the VCE, the final application is shown in Figure 5-15 on page 268.

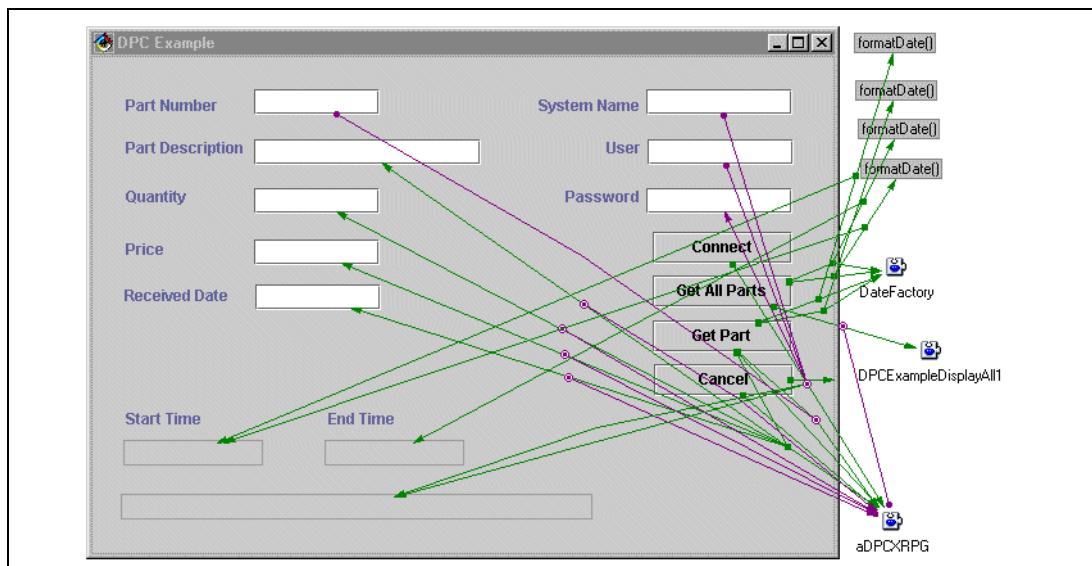


Figure 5-15 Completed application

That completes the processing to connect to the iSeries server and to call an iSeries RPG program. To complete the application, we supply a DPCEexampleDisplayAll bean to handle retrieving all parts from the database.

We built this entire application in the VisualAge for Java Visual Composition Editor. We did not write a single line of code. We used the Distributed Program Call bean that we generated using the ET/400 SmartGuide to do all the work. An additional benefit of the DPCXRPG bean is that it can easily be made available for others to use in their applications.

## 5.4 SmartGuide to convert iSeries display files to Java

The Convert Display File SmartGuide allows you to quickly convert existing 5250 screens to Java. It converts existing iSeries display file records to Java Swing code. You can use the generated Java code as a bean in the Visual Composition Editor, or as runnable code in an application.

All classes are generated to conform to the JavaBean specifications. The name of the classes that are generated are based on the display file records you selected to convert.

For each display file record, four classes are added to the selected package:

- ▶ Converted record name
- ▶ Converted record nameEventMulticaster
- ▶ Converted record nameListener
- ▶ Converted record nameWindowListener

A log file, importSG.log, saves this list of classes so you can quickly identify the converted records.

The output usage is:

- ▶ As is — Manually code to the Java generated code
- ▶ In the Visual Composition Editor as visual parts

The SmartGuide allows you to select which DDS file you want to use as input for the conversion. In Figure 5-16, we select DSPPARTS.

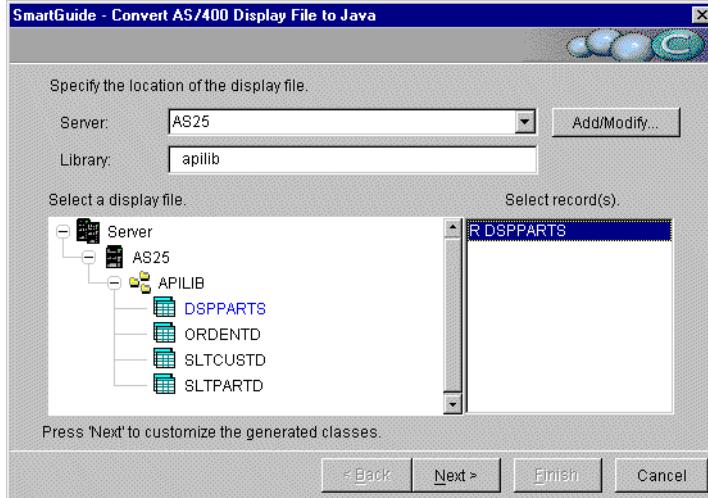


Figure 5-16 Convert DDS SmartGuide

Figure 5-17 shows how the DSPPARTS file appears when used in a 5250 application running on an iSeries server.

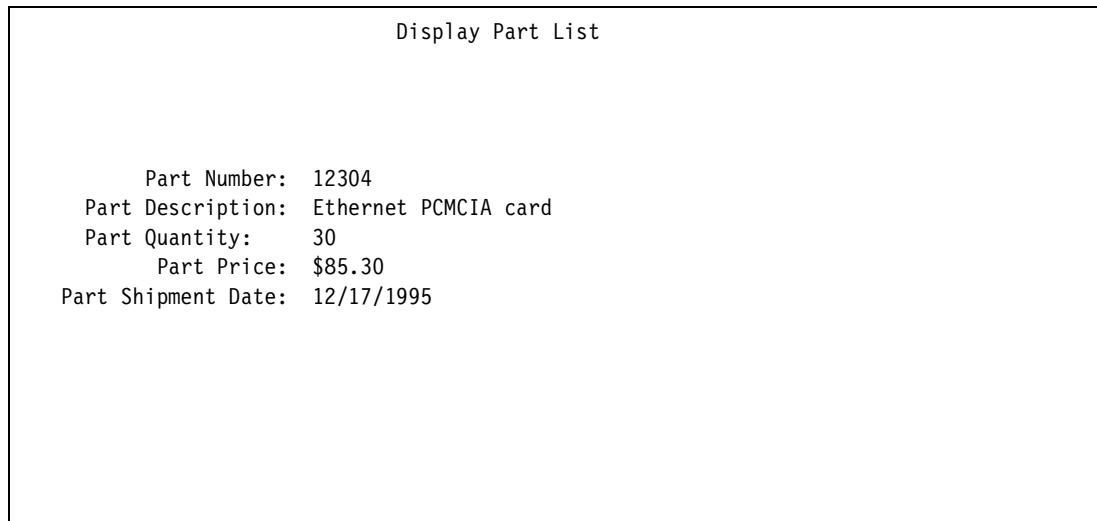


Figure 5-17 DSPPARTS file in a 5250 application

Example 5-2 shows the source for the DSPPARTS display file. We use it to create a Java Swing class that is comparable.

#### Example 5-2 DSPPARTS Display File Source

```
DSPSIZ(24 80 *DS3)
R DSPPARTS
    2 32'Display Part List'
    7 11'Part Number:'
        DSPATR(HI)
PARTNO      R          B  7 25REFFLD(PARTR/PARTNO APILIB/PARTS)
```

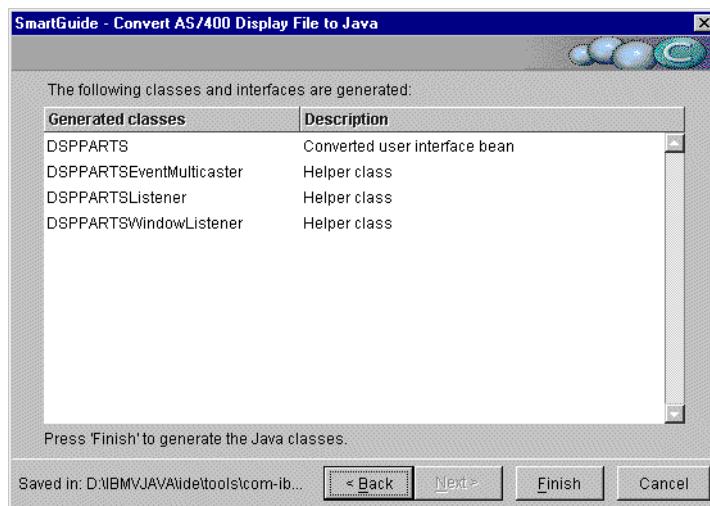
```

          DSPATR(CS)
          EDTCDE(3)
8 6'Part Description:'
PARTDS    R     B 8 25REFFLD(PARTR/PARTDS APILIB/PARTS)
9 6'Part Quantity:'
PARTQY    R     B 9 25REFFLD(PARTR/PARTQY APILIB/PARTS)
          EDTCDE(4)
10 12'Part Price:'
PARTPR   R     B 10 25REFFLD(PARTR/PARTPR APILIB/PARTS)
          EDTCDE(3 $)
11 4'Part Shipment Date:'
PARTDT   R     B 11 25REFFLD(PARTR/PARTDT APILIB/PARTS)

```

---

As shown in Figure 5-18, the SmartGuide generates a Java class for the DDS file and several helper classes.



*Figure 5-18 DDS Conversion SmartGuide*

Figure 5-19 shows how the converted output appears as a Java application. The created JavaBean can be used as part of an application or as a visual component in a visual builder such as the VisualAge for Java IDE.

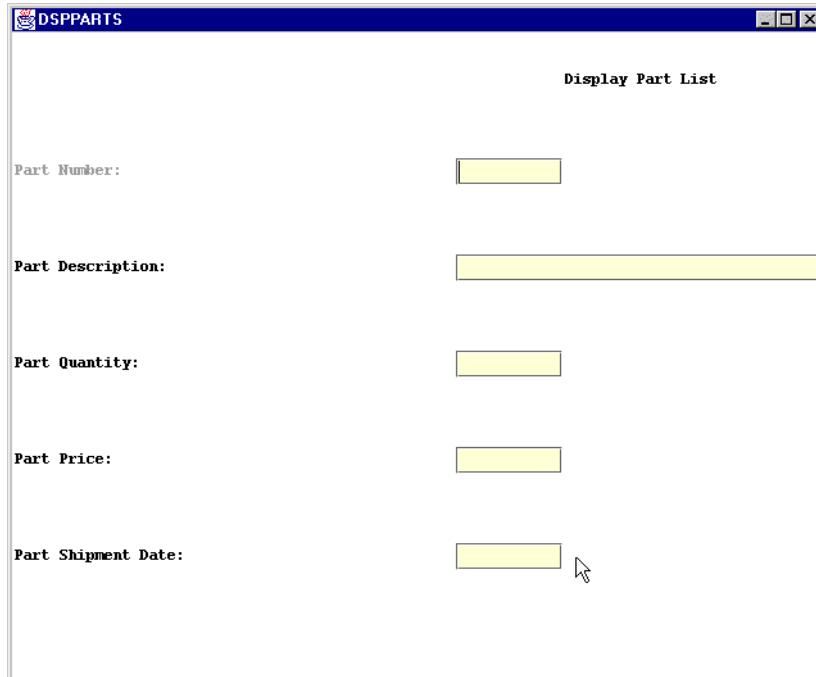


Figure 5-19 Converted DDS file

The convert DDS SmartGuide allows you to quickly convert DDS source files to the Java Swing format. The resulting JavaBean can be used as part of a Java application. The SmartGuide uses Swing, where possible. If not, it uses the com.ibm.ivj.et400.util package to deal with the rest of the fields. For example, a subfile extends com.ibm.ivj.et400.util.AS400VisualSubfile.

## 5.5 iSeries beans

ET/400 provides three groups of JavaBeans, which can be used to extend iSeries Java applications with iSeries-unique features:

- ▶ JFormatted beans
- ▶ Data file Utility (DFU) beans
- ▶ Object List beans

### 5.5.1 JFormatted beans

The JFormatted beans include a set of utility classes that extends the support of code to convert iSeries fields and attributes and to provide edit code, edit word, formatting, and verification capabilities. Table 5-4 on page 272 lists the JFormatted beans.

Table 5-4 JFormatted beans list

Beans palette icon	Bean name	Functions
	com.ibm.ivj.et400.util.ASFieldModel	Sets the defaults for the non-iSeries applications.
	com.ibm.ivj.et400.util.DefaultFieldModel	Sets the defaults for the iSeries specific attributes. It is default field model that is used by the JFormatted beans.
	com.ibm.ivj.et400.util.JFormattedComboBox	Has all the behaviors of a Swing JComboBox. Can be used to implement the VALUES feature that limits the field to preset values in iSeries Server.
	com.ibm.ivj.et400.util.JFormattedLabel	A label that allows you to define a field model, which can specify a formatter, a validator, and data attributes.
	com.ibm.ivj.et400.util.JFormattedTable	Same behavior as the Swing JTable. But, as an extra feature, it will recognize the JFormattedTableColumn. Supports edit code, edit word, formatting, and verification capabilities.
	com.ibm.ivj.et400.util.JFormattedTableColumn	Same behavior as Swing TableColumn, but with some additional features. Supports edit code, edit word, formatting, and verification of iSeries data, and set FieldModel and so on.
	com.ibm.ivj.et400.util.JFormattedTextField	Same behaviors as the Swing JTextField. However, it allows you to define field model; to specify a formatter, a validator, a keystroke verifier, and data attributes; and so on.
	com.ibm.ivj.et400.util.JFormattedComboBoxCellEdit	Table cell editor component that supports formatting and verification of iSeries data. Should be used with a JFormattedTableColumn bean.
	com.ibm.ivj.et400.util.JFormattedLabelCellRenderer	Table cell renderer component that supports formatting of iSeries data. Should be used with JFormattedTableColumn.
	com.ibm.ivj.et400.util.JFormattedTextFieldCellEditor	Table cell editor component that supports edit code, edit word, formatting, and verification of iSeries data. Should be used with JFormattedTableColumn.

### JFormatted beans example

The objective of section is to take you through the steps to build an application with JFormatted beans. It demonstrates how to use the JFormattedTextField and JFormattedComboBox classes and features.

The finished application looks like the example in Figure 5-20.

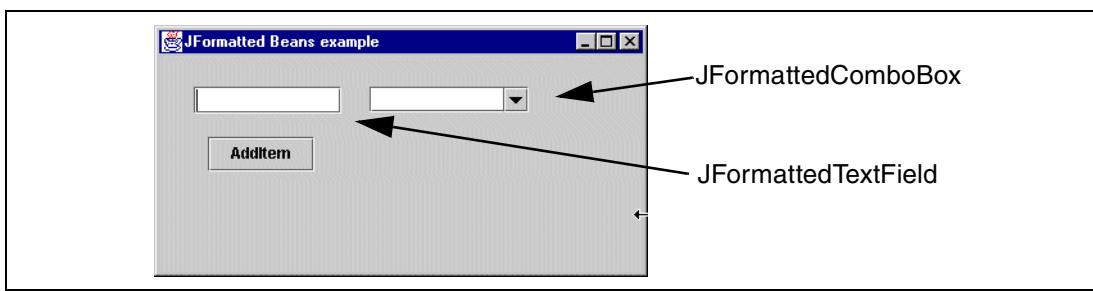


Figure 5-20 JFormatted beans example

1. Create a package such as **iSeriesBeanExample** under a project such as **ET400Example**.
2. Add a new class using the Create Class SmartGuide, and specify the following information:
  - **JFormattedBeanExample** in the Class Name field
  - **javax.swing.JFrame** in the Superclass field
  - Select the **Compose the class visually** and **Browse the class when finished** check boxes.
3. Click **Finish**. The SmartGuide closes and the Visual Composition Editor opens. Change the properties **title** of the **JFrame** to **JFormatted beans example**.
4. Make sure that the **AS400 ToolBox** category from the category selection is selected (Figure 5-21).

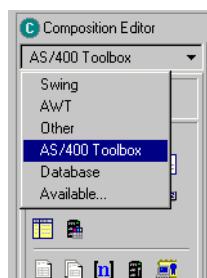


Figure 5-21 Selecting the AS/400 Toolbox category

5. Select the **JFormattedTextField** bean from the Beans palette. Drop it on the upper area of the frame, and then resize the **JFormattedTextField** bean.
6. Double-click the **JFormattedTextField** bean to open the Properties window.
  - a. Click the field next to **fieldModel**. Click the list to open the FieldModel Implementor window. See Figure 5-22 on page 274.

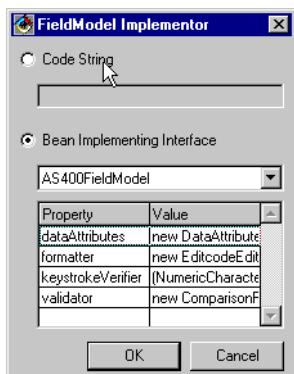


Figure 5-22 The FieldModel Implementor window

- b. Click the value next to the **formatter** property. Click the list to open the Formatter Implementor window (Figure 5-23).

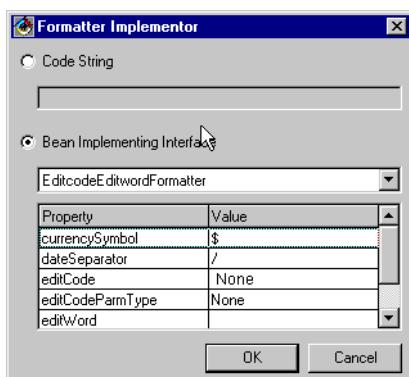


Figure 5-23 The FieldModel Implementor formatter window

- c. Click the value next to the **editCode** property. Then click the list. Select **1** from the EditCode window (Figure 5-24). Click **OK**.

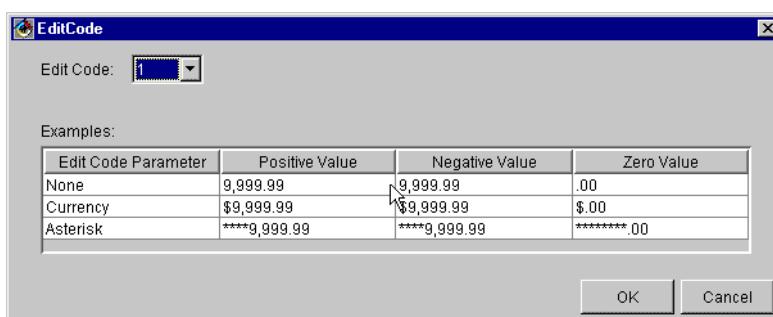


Figure 5-24 Using the FieldModel Implementor formatter to set the editCode

- d. In the Formatter Implementor window, click the value next to the **editCodeParmType** property. Then click the list and select **Asterisk** (Figure 5-25). Click **OK** to return to the FieldModel Implementor window.

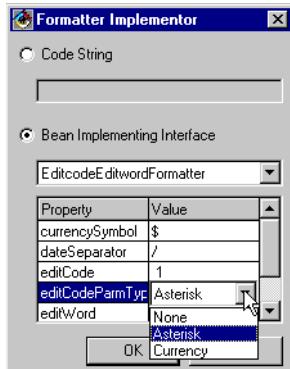


Figure 5-25 Using the Formatter Implementor to set the editCodeParmType

- e. In the FieldModel Implementor window, click the value next to the **dataAttributes** property. Click the list to open the Attributes Implementor window.
- f. Click the value next to the **dataType** property. Then click the list and select **Numeric**. See Figure 5-26. Click **OK**, and then click **OK** again to close the FieldModel Implementor window.

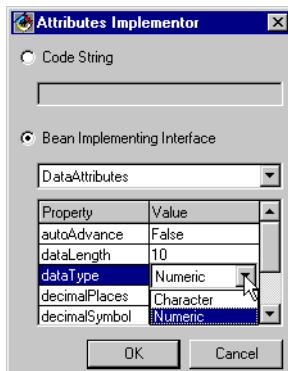


Figure 5-26 Using the Attributes Implementor to set the dataType

7. From the Beans Palette list, select and drop a **JFormattedComboBox** bean onto the upper area of the frame to the right of the JFormattedTextField. Resize the **JFormattedComboBox** bean. See Figure 5-27.

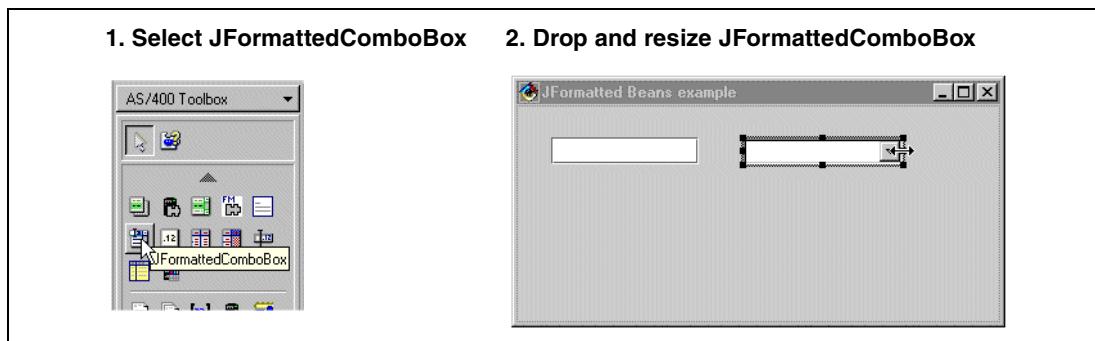


Figure 5-27 Creating the JFormattedComboBox bean

8. Double-click the **JFormattedComboBox** bean to open the Properties window.
- a. Click the field next to **fieldModel**. Click the list to open the FieldModel Implementor window (see Figure 5-22 on page 274).

- b. Click the value next to the **formatter** property. Click the list to open the Formatter Implementor window (see Figure 5-23 on page 274).
- c. Click the value next to the **editCode** property. Then click the list and select **A**. Click **OK**. See Figure 5-28.

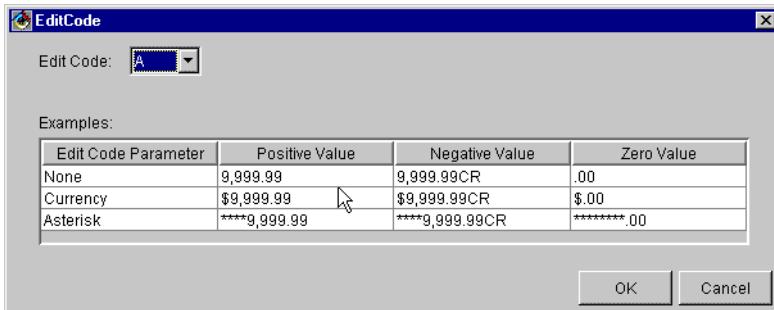


Figure 5-28 Using the Formatter Implementor to set the editCode

- d. Back in the FieldModel Implementor formatter window (Figure 5-23), click the value next to the **editCodeParmType** property. Then click the list and select **Currency** (see Figure 5-29). Click **OK**.

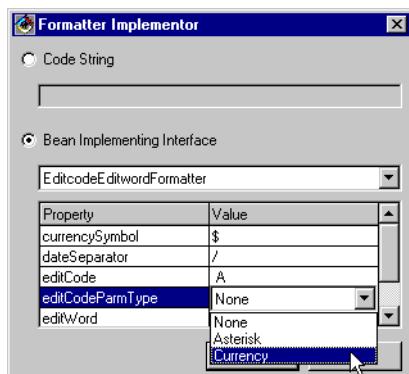


Figure 5-29 Using the Formatter Implementor to set the editCodeParmType

- e. In the FieldModel Implementor formatter window, click the value next to the **dataAttributes** property. Click the list to open the Attributes Implementor window.
- f. Click the value next to the **dataType** property. Then click the list and select **Numeric** (see Figure 5-30). Click **OK**, and then click **OK** again to close the FieldModel Implementor window.

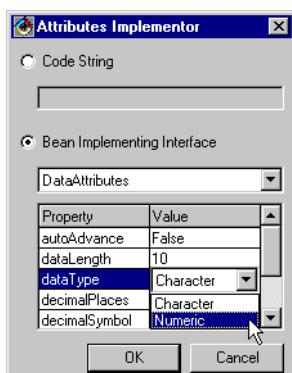


Figure 5-30 Using the Attributes Implementor to set dataType

9. Select the **Swing** category from the Beans Palette list. Select the **JButton** bean and drop it onto the frame below the JFormattedTextField.
  - a. Right-click the **JButton** bean and select **Properties**.
  - b. Click the field next to **text** and type **AddItem**.
  - c. Click the field next to **beanName** and type **JButtonAddItem**.
  - d. Close the properties window.
10. To save your work, select **Bean-> Save Bean**.
11. Now we make the connections:
  - a. Select the **JButtonAddItem** button and right-click. In the pop-up menu that appears, select **Connect -> actionPerformed**.
  - b. Position the mouse pointer over the **JFormattedComboBox** bean and click to select it. In the pop-up menu that appears, select **Connectable Features**.
  - c. The End connection to window opens. Select the **addItem(Object)** method (Figure 5-31). Click **OK**. Now you see a dashed dark green line, which means an incomplete connection.

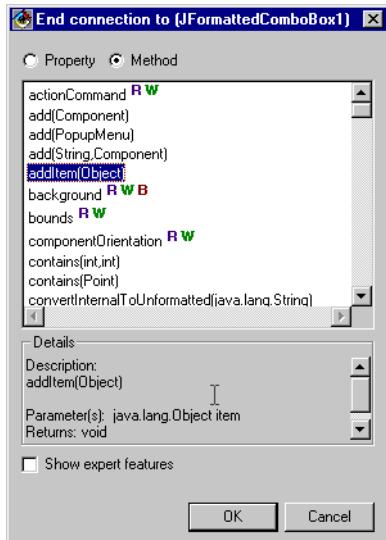


Figure 5-31 Connecting the Add Item button action to the JFormattedComboBox

- d. Right-click the dashed dark green line. In the pop-up menu that appears, select **Connect -> Item**. See Figure 5-32.

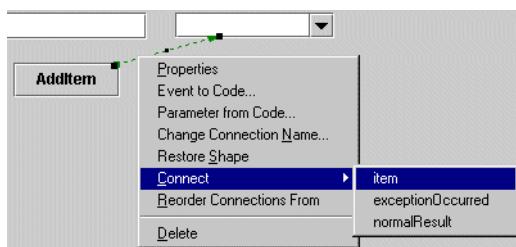


Figure 5-32 Adding the first connection from the Add Item button to the JFormattedComboBox

- e. Position the spider over the **JFormattedTextField** bean, and click to select it. In the pop-up menu that appears, select **Connectable Features**.

- f. The End connection to window (Figure 5-33) opens. Select the **getUnformattedText()** method. Click **OK**.

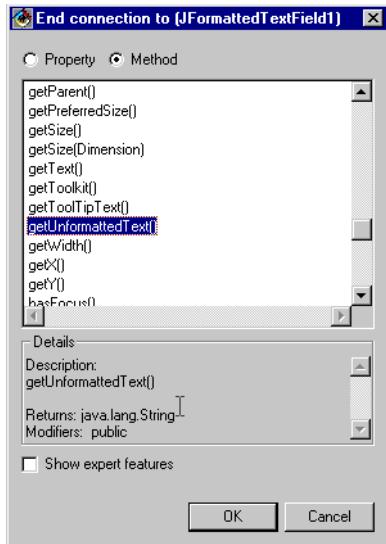


Figure 5-33 Completing the first connection to the JFormattedComboBox

- g. Select the **JButtonAddItem** button again, and then right-click. In the pop-up menu that appears, select **Connect -> actionPerformed**.
- h. Position the mouse pointer over the **JFormattedComboBox** bean and click. In the pop-up menu that appears, select **Connectable Features**.
- i. The End connection to window opens. Select the **repaint()** method. See Figure 5-34. Click **OK**.

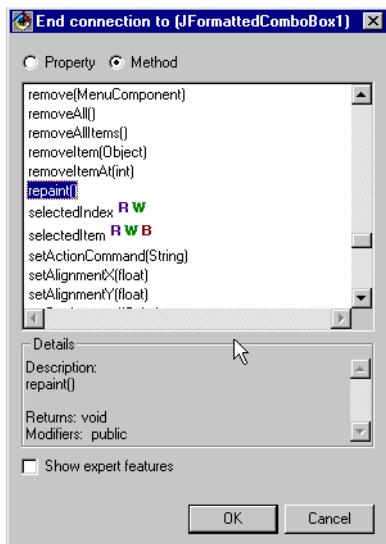


Figure 5-34 Adding a second connection to the JFormattedComboBox to repaint it

- j. After completing the connections, the frame looks like the example in Figure 5-35.

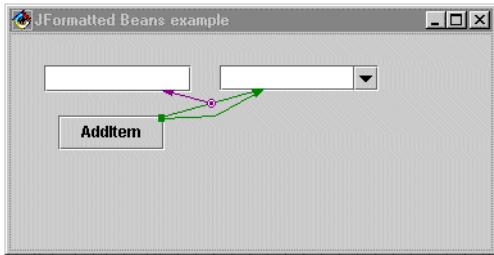


Figure 5-35 JFormatted beans example

#### 12. Testing the completed application

- Set the classpath using **Bean->Run->Check Classpath ->Compute Now.**
- Click **OK** to return to the VCE.
- Select **Run** from the toolbar. A window opens displaying the GUI just created.
- Select the **text field** and enter a number. The number will be formatted when you tab to the next field using the formatter defined in the JFormattedTextField bean fieldModel.
- Click the **AddItem** button. The number you entered will be added to the JFormattedComboBox. The number will be formatted according to the formatter defined in the JFormattedComboBox bean fieldModel.

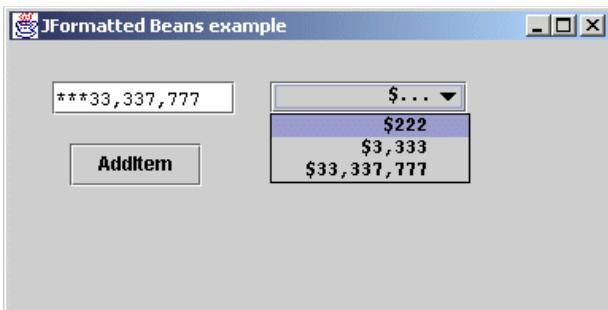


Figure 5-36 Running the application

### 5.5.2 Data File Utility (DFU) beans

The DFU beans include a set of classes that allow you to access iSeries tables. They can map Swing components with data fields in iSeries tables. The data retrieved is automatically displayed in the Swing component fields. Any data changes in the Swing components can be reflected in the database. The Swing components can be a Convert Display File generated bean, JFormattedTable, or any user-defined GUI screen.

Table 5-5 DFU beans list

Beans palette icon	Bean name	Functions
	com.ibm.ivj.et400.util.FormManager	Used for connecting a form to a database file or record format by associating JComponents in the form to database field names of the same name.
	com.ibm.ivj.et400.util.ListManager	Used for connecting a JComponent that holds multiple rows of data to a database file. Supports multiple records.

Beans palette icon	Bean name	Functions
	com.ibm.ivj.et400.util.RecordIOManager	Used for record manipulation, such as open/close Database file, retrieval of records from database or record addition, updating, or deleting.

### DFU beans example

The objective of this section is to build an application, which uses DFU beans, JFormatted beans, and beans generated by the Convert Display File function. You can complete this simple database operation application without writing a single line of code.

The finished example looks like the display in Figure 5-37.

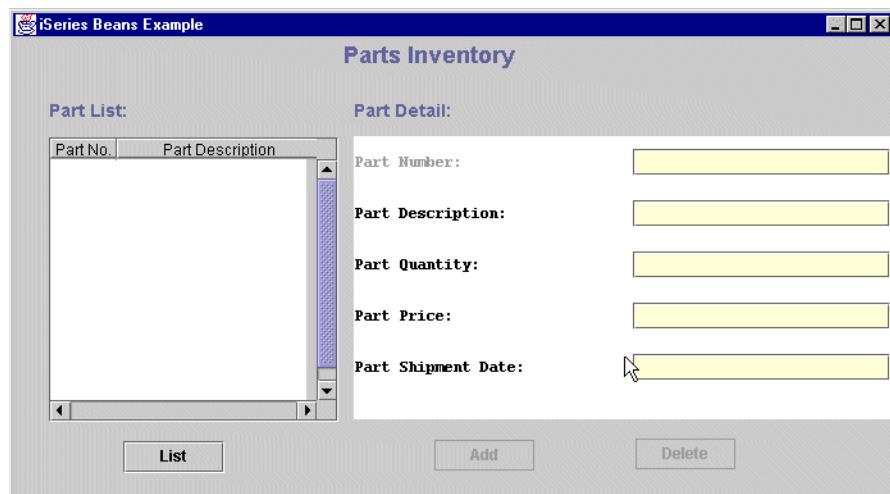


Figure 5-37 DFU beans example

With this application, you retrieve records from an iSeries table by clicking the **List** button. You can select one record in the list, and the record's detail information will be displayed in the **Part Detail** panel on the right. You can use the **Delete** button to delete a record from the database and add a new record to the database by using the **Add** button. The completed example is available from the ITSO web site. We provide the steps used to create it:

1. Create a class named DFUBeanExample that extends JFrame (Figure 5-38). Click **Finish** and resize the JFrame. Set the **title** property of the JFrame to iSeries Beans example.

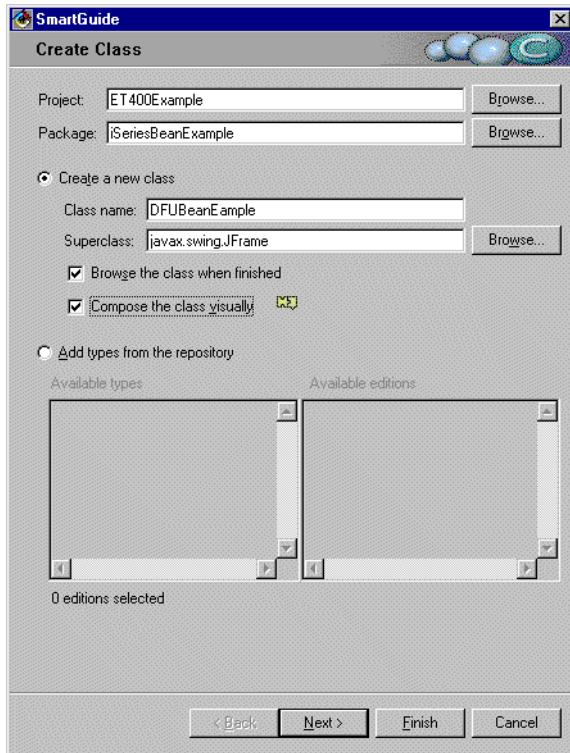


Figure 5-38 Creating the DFUBeanExample class

2. Use **JLabel** beans to add three titles: Parts Inventory, Part List, and Part Detail.
  - a. Select **JLabel** from the Swing category bean. Drop it onto the upper area of the **JFrame**.
  - b. Double-click the **JLabel** bean. On the Properties window, change the beanName, text, and horizontalAlignment values as shown in Figure 5-39 on page 282. Use the re-size handles to increase the width of the label bean to accommodate the text. Leave the Properties window open.

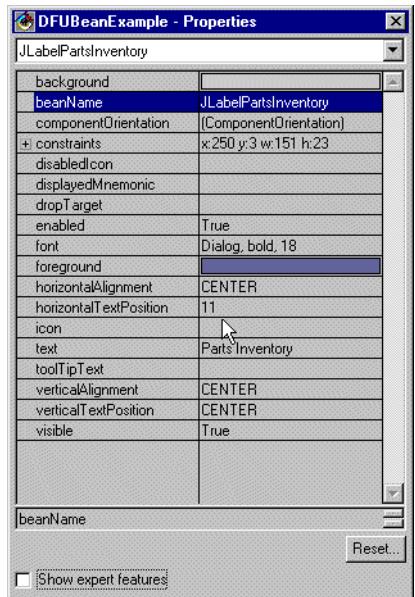


Figure 5-39 Setting the `JLabel` properties

- Repeat the above steps to add the other two titles, Part List with `JLabelInventoryList` as the beanName, and Part Detail with `JLabelPartDetail` as the beanName. See Figure 5-40.



Figure 5-40 Creating three `JLabels`

- Create the Part List with one `JFormattedTable` and two `JFormattedTableColumn` beans.

The `JFormattedTable` is used to display and edit iSeries database records. Each record format field is defined using the `JFormattedTableColumn` bean.

- Select the AS/400 Toolbox category from the Beans Palette list.
- From the Beans Palette list, select **JFormattedTable** and drop it below the Part List **JLabel** at the desired position. Resize it. See Figure 5-41.

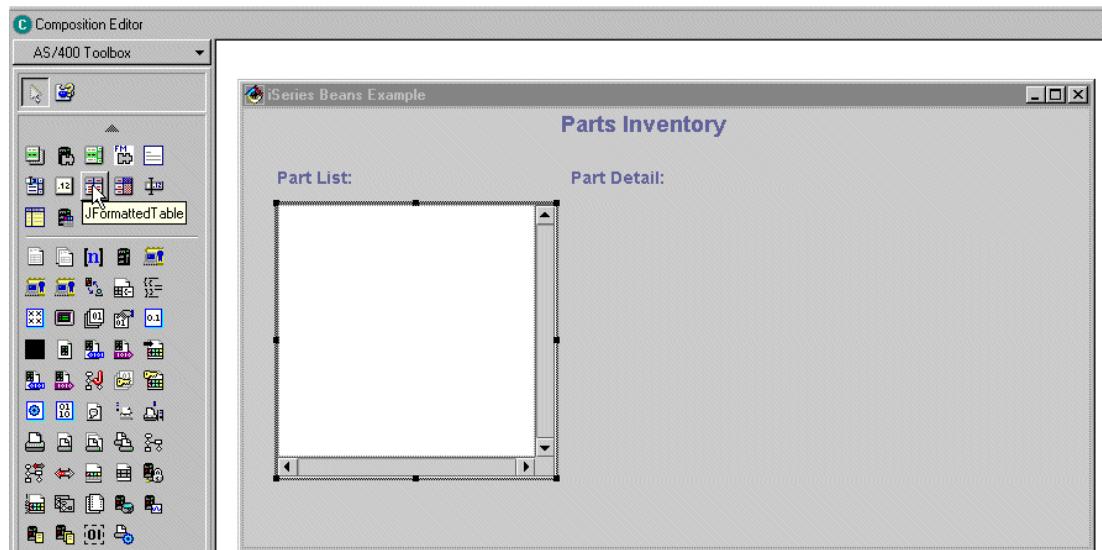


Figure 5-41 Creating the JFormattedTable

- c. Select the **JFormattedTableColumn** bean and drop it on the JFormattedTable bean.
- d. Repeat the above step to add a second JFormattedTableColumn bean.
- e. You may want to resize the two **JFormattedTableColumn** beans so that they fit into the table (Figure 5-42).

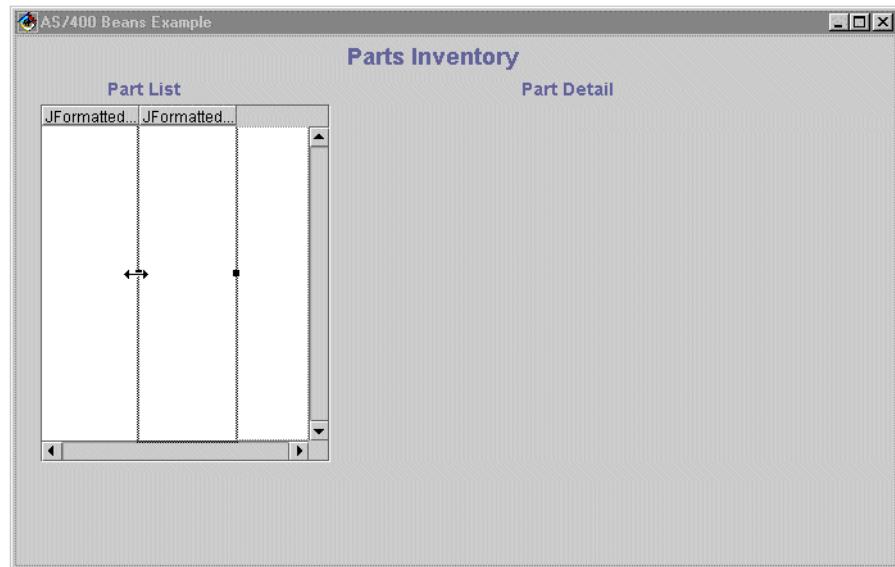


Figure 5-42 Adding the JFormattedTableColumn beans

- f. Right-click the first **JFormattedTableColumn** bean and select **Properties**.
- g. Click the field next to **identifier**, and enter **PARTNO** (complete with quotation marks).
- h. Click the field next to **headerValue** and enter Part No.
- i. Click the field next to **cellEditable** and change it to "False". See Figure 5-43 on page 284.

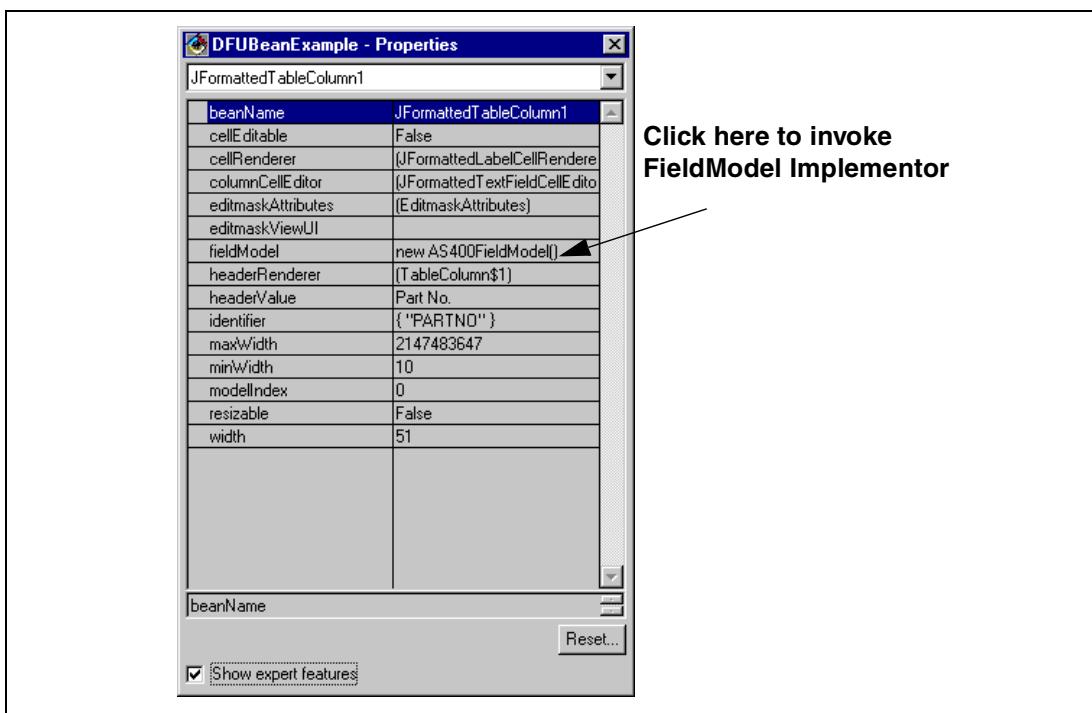


Figure 5-43 Setting the *JFormattedTableColumn* properties

- Select **fieldModel**. Click the secondary window button to invoke the FieldModel Implementor. See Figure 5-44.

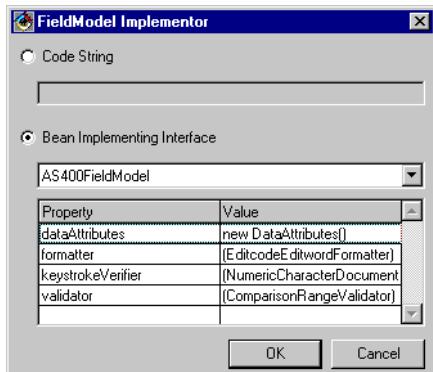


Figure 5-44 Using the *JFormattedTableColumn* FieldModel Implementer

- Select the field next to **dataAttributes** and click the secondary window button to invoke the Attributes Implementor. Set the dataLength value as 5. See Figure 5-45.

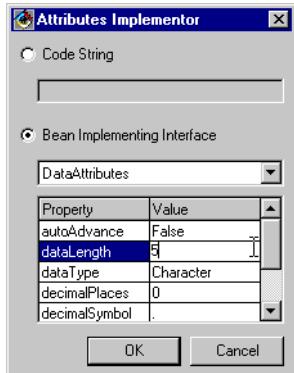


Figure 5-45 Setting the dataLength attribute

- I. Click **OK** to close the Attributes Implementor. Then click **OK** again to close the FieldModel Implementor.
- m. Repeat the above steps to set the second JFormattedTableColumn bean properties. Set identifier as “PARTDS”, headerValue as **Part Description**, cellEditable as **False**, and dataLength as 25.
4. Change the DSPPARTS JavaBean.
  - a. We use the DSPPARTS bean discussed earlier. See 5.4, “SmartGuide to convert iSeries display files to Java” on page 268, to display the record details. It was created using the ET/400 Convert Display File utility. We need to make some modifications to it.
  - b. As shown in Figure 5-46, select the **DSPPARTS** class in the ConvertDSFP package, and open it in the VCE.

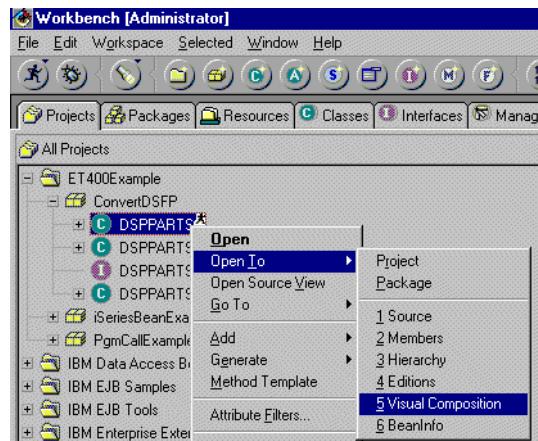


Figure 5-46 Opening the DSPPARTS class

- c. Delete the Display Part List label, and change the five JFormattedTextFields' **beanName** properties to **PARTNO**, **PARTDS**, **PARTQY**, **PARTPR**, and **PARTDT** respectively. Resize the beans (see Figure 5-47 on page 286). Save your work by selecting **Bean-> save Bean**.

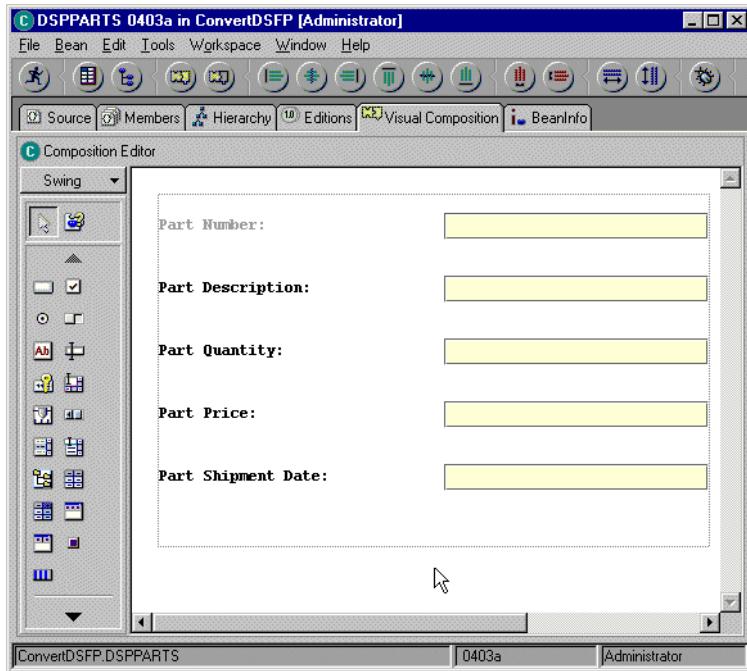


Figure 5-47 Modifying the DSPPARTS JavaBean

- Create the PartDetail bean using the ConvertDSPF class.

Return to the DFUBeanExample Visual Composition Editor window, and add the DSPPARTS bean to it. To do this, select from the Beans Palette to open the **Choose Bean** dialog box (Figure 5-48). Click **OK**, and drop the bean below the Part Detail JLabel bean.

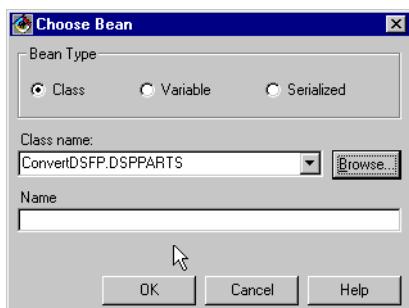


Figure 5-48 Creating the PartDetail bean

- Create the ListManager, FormManager, and RecordIOManager beans.

- Select the AS/400 Toolbox category from the Beans Palette list, if it has not already been selected. Select **ListManager**, from the Beans Palette list, and drop it on the free-form surface.
- Repeat the above step to create both the FormManager and RecordIOManager beans. See Figure 5-49.

- Add the List, Add, and Delete buttons.

- Select the **JButton** bean from Swing category and drop it below the JFormattedTable bean.
- Change its text properties to **List**.

- c. Repeat these steps to add two more buttons below the ConvertDSPF bean. Set the “text properties” to **Add** and **Delete** respectively. Set “enabled” to **False**. See Figure 5-49.

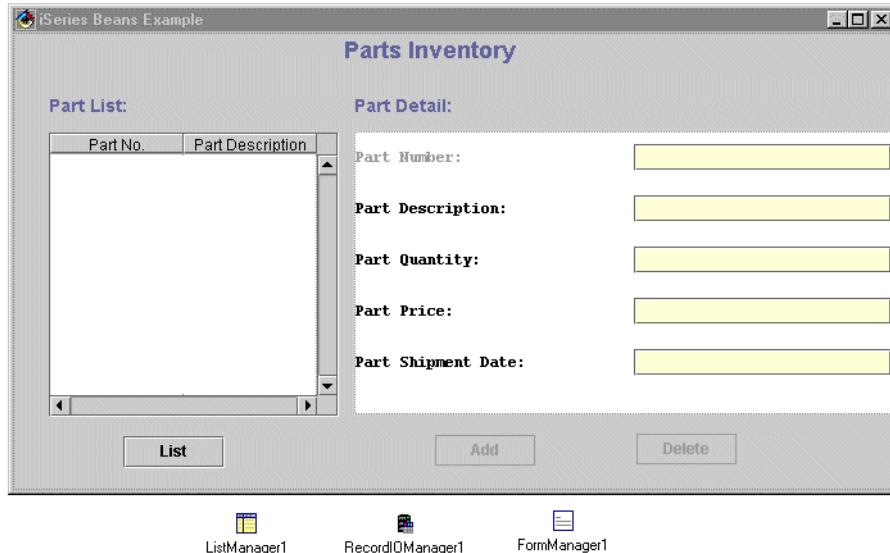


Figure 5-49 *iSeriesBeansExample* components

7. Set the properties for the RecordIOManager bean:

- Double-click the **RecordIOManager** bean to display the properties setup window.
- Set fileName as **parts**, library as **apilib**, server as **your-iSeries-server-name**. See Figure 5-50.

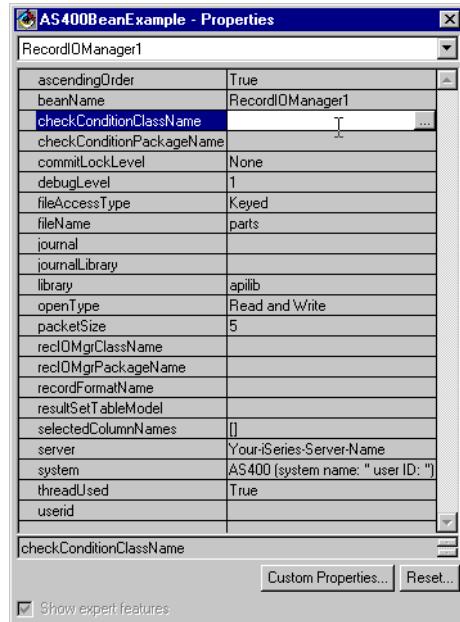


Figure 5-50 Setting the RecordIOManager properties

- c. On the Properties dialog (Figure 5-50), click the **Custom Properties...** button to open the RecordIOManagerCustomizer window. Complete the RecordIOManager properties settings. You can specify the user name and password for the iSeries connection here. See Figure 5-51 on page 288.

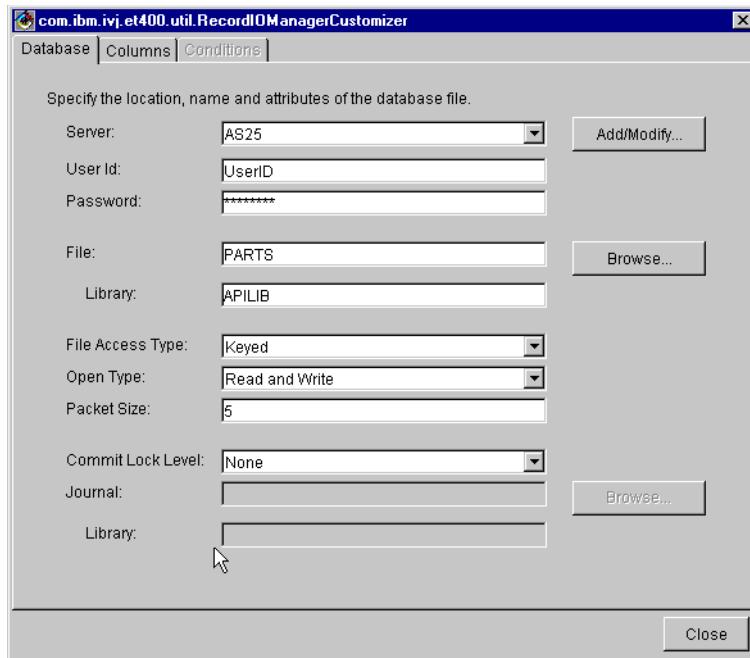


Figure 5-51 RecordIOManagerCustomize window

#### 8. Modify the JFormattedTable selectionModel for Part Detail display updates:

When you click a row in the Part List table (in the running application), you want the record of the selected part number to be displayed in the Part Detail panel. When a value is changed in the selectionModel of the table, you want to run the readRecord(int) method in the FormManager object. To access the selectionModel of the JFormattedTable, you need to “tear off” (using the Tear-Off Property feature) the selectionModel property from the JFormattedTable.

- In the Visual Composition Editor, select **Tools-> Beans List**.
- In the Beans List window, right-click the **ScrollPaneTable** object and select **Tear-Off Property** from the pop-up menu. See Figure 5-52.

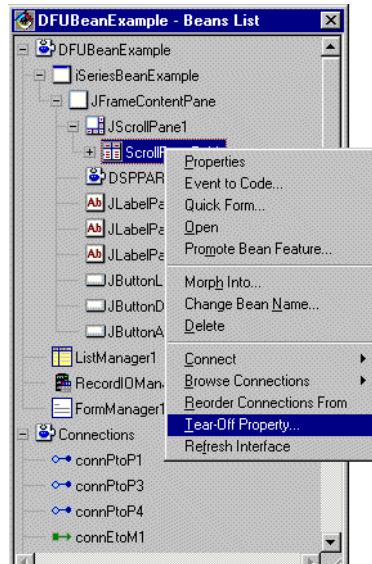


Figure 5-52 Selecting the bean from which to tear-off a property

- c. In the Tear-Off Property Dialog, select **selectionModel(ListSelectionModel)** (Figure 5-53). Click **OK**.

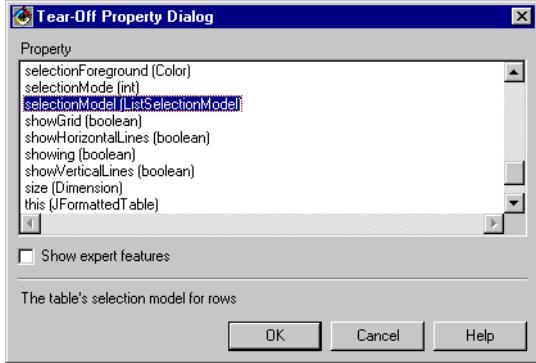


Figure 5-53 Selecting the property to tear off

A **selectionModel** object is created in the free-form surface with a connection to the **ScrollPaneTable**. See Figure 5-54.

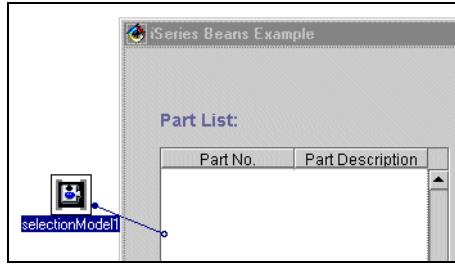


Figure 5-54 The torn-off property

9. Making the connections to create the DFU bean display containers:

To specify the **displayContainer** for the **ListManager** and **FormManager** beans, we connect the **this** property of the **JScrollPane** to the **displayContainer** property of the **ListManager** bean. We also connect the **this** property of the **ConvertDSPF** bean to the **displayContainer** property of the **FormManager** bean. See Figure 5-55.

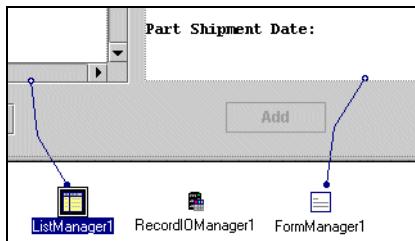


Figure 5-55 Connections for the displayContainer property

10. Specify the **RecordIOManager** for the **ListManager** bean and the **FormManager** bean:

Connect the **recordIOManager** property of the **ListManager1** and the **FormManager1** bean to the **this** property of the **RecordIOManager1** bean respectively. See Figure 5-56 on page 290.

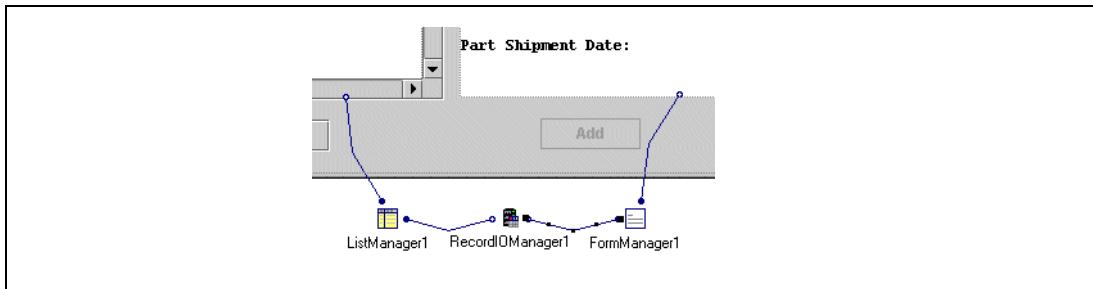


Figure 5-56 Specifying the RecordIOManager for the ListManager and FormManager beans

**Note:** The ListManager and FormManager beans are both using the same instance of RecordIOManager.

11. Make the connections to modify the JFormattedTable selectionModel for the Part Detail display updates:

When you select a row in the JFormattedTable (in the running application), the selectionModel of the table will launch a valueChanged event. When this occurs, it will call the readRecord(int) method of the FormManager bean to read a record from the database, with the specified record number as the parameter. The record number is retrieved from the ListManager bean with the getRecordNumber(int) method, where the parameter is the selected row number in the table.

- a. Select the **selectionModel** bean and right-click. In the pop-up menu that appears, select **Connect-> valueChanged**.
- b. Position the spider over the FormManager bean, and click to select it. In the pop-up menu that appears, select **Connectable Features**. The End connection to window opens. Select the **readRecord(int)** method. Click **OK**.
- c. At this stage, the connection is incomplete with a dashed line. Right-click the dashed line and select **Connect-> recordNumber**. Position the mouse spider over the ListManager bean and click. In the pop-up menu that appears, select **Connectable Features**. The End connection to window opens. Select the **getRecordNumber(int)** method and click **OK**.
- d. Select **Tools -> Beans List** from the menu to open the Beans list window. Select the incomplete connection and right-click it. Select **Connect-> row**. Move the spider over the **ScrollPaneTable** and click. In the pop-up menu that appears, select **selectedRow**. See Figure 5-57.

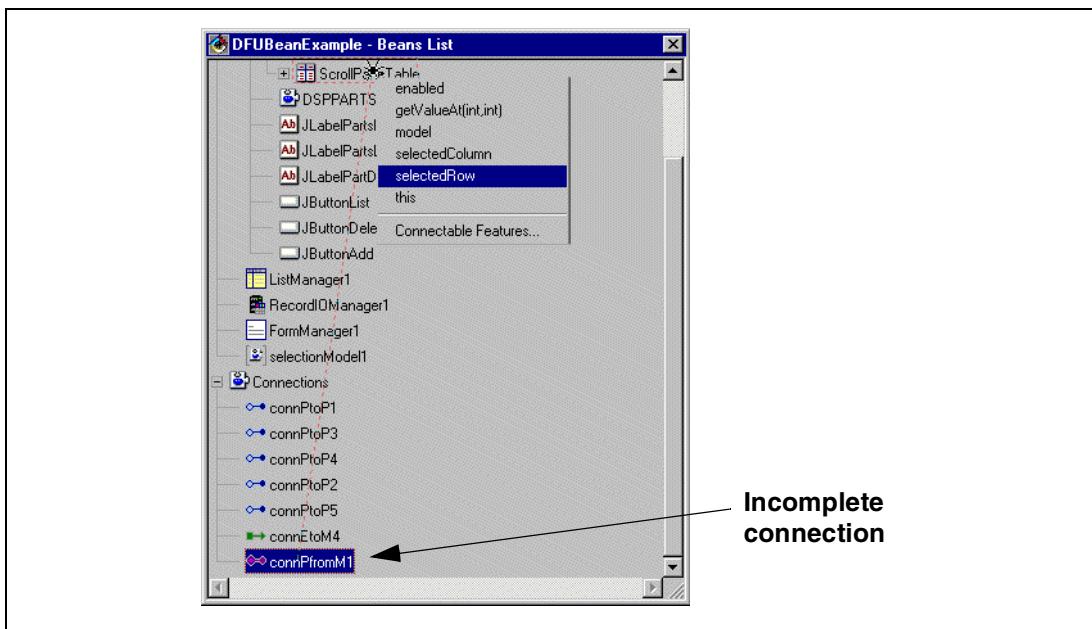


Figure 5-57 Complete connection with the selectedRow property of the ScrollPaneTable

e. Close the Beans List window.

#### 12. Complete the button connections:

To list all records in the JFormattedTable, we connect the ActionPerformed event of the List button to the readAllRecords() method of the ListManager bean. See Figure 5-58.

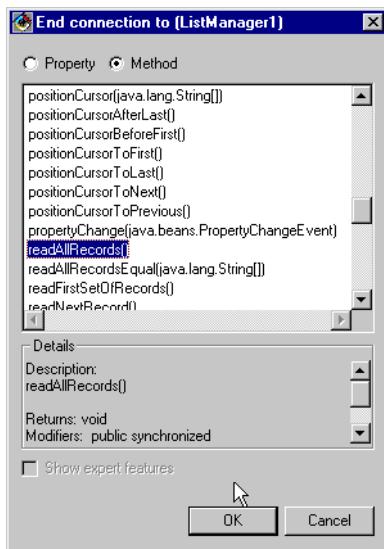


Figure 5-58 Adding a connection for the List button

- To add a new record, we connect the ActionPerformed event of the Add button to the addRecord() method of the FormManager bean. When the application is running and the Add button is clicked, FormManager will add a record to the database file defined in the RecordIOManager1 bean, with the values currently in the ConvertDSPF bean.
- To delete a record, we connect the ActionPerformed event of the Delete button to the deleteRecord() method of the FormManager bean. When the application is running

and the Delete button is clicked, FormManager will delete the record from the database file defined in the RecordIOManager bean. See Figure 5-59.

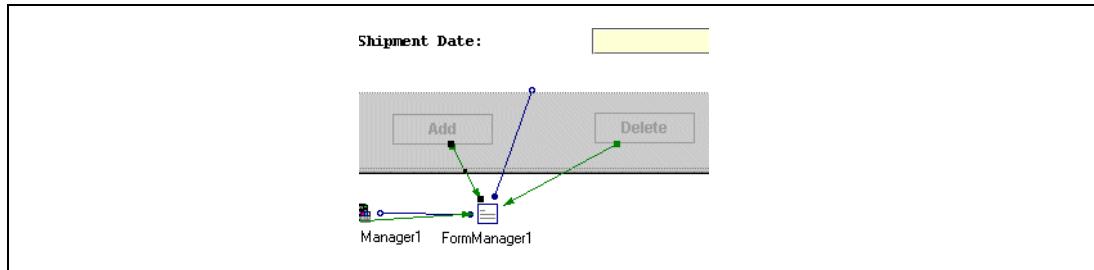


Figure 5-59 Adding connections for the Add and Delete buttons

13. Make the connections to refresh the Part List and the Part Detail:

- To refresh the part list after a record is added to the database file, you need to implement a `readAllRecords()` method on the ListManager bean. Select the connection that was made for `addRecord()` and right-click. In the pop-up menu that appears, select **Connect -> normalResult** and connect to the `readAllRecords()` method of the ListManager bean.
- To enable clearing the form after a record is added, select the connection that was made for `addRecord()`, and right-click. In the pop-up menu that appears, select **Connect -> normalResult** and connect it to the `clearAllData()` method of the FormManager1 bean. Click **OK**. See Figure 5-60.

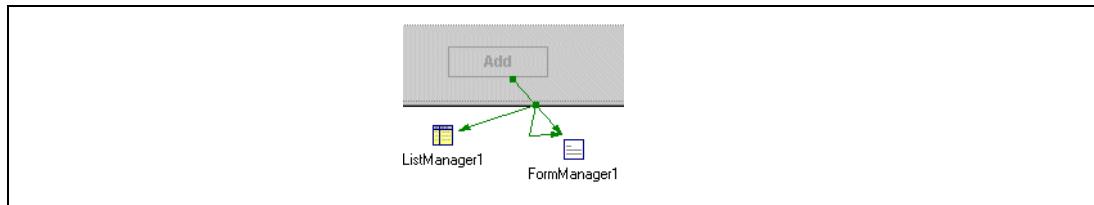


Figure 5-60 Adding connections to refresh Part List and Part Detail

- Repeat the above steps for the delete operation.

14. Test your completed application:

- The application should now look like the example in Figure 5-61. Select **Bean -> Save Bean** to save the application that you created.
- Choose **Bean->Run->Check Classpath ->Compute Now** to set the classpath.
- Click **OK** to return to the VCE.
- Select **Run** from the toolbar. A window opens displaying the GUI you created. Click the **List** button and an iSeries Sign On window appears. Enter your user ID and password.
- Click the **List** button to retrieve the data from the iSeries database.
- Select a record from the list. The record details should be displayed in the Part Detail panel. You can click the **Delete** button to delete a record and click the **Add** button to add a new part record to the database.

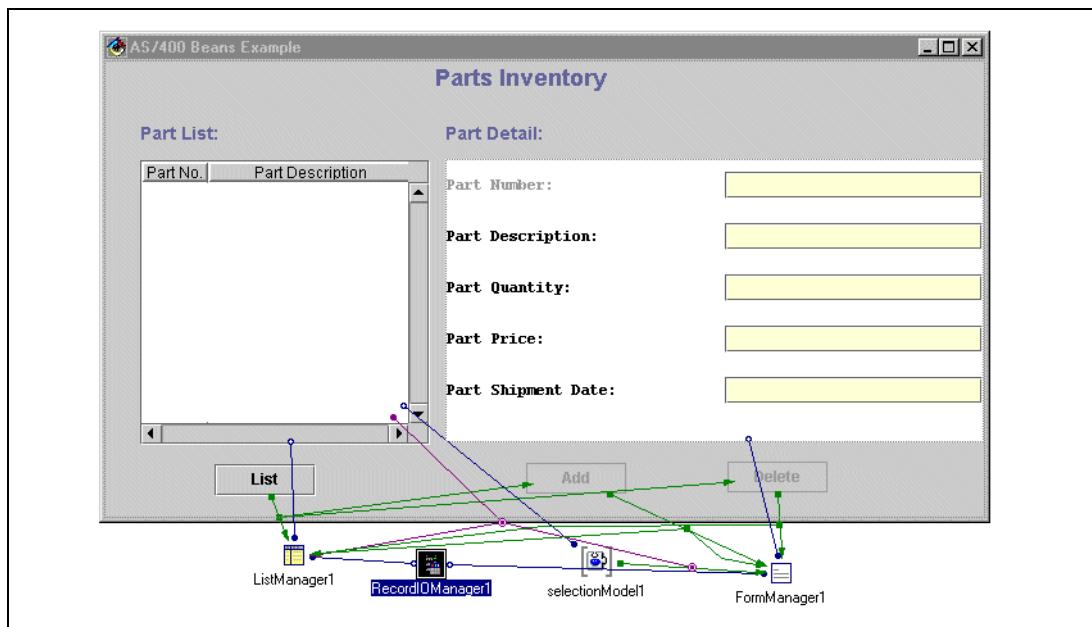


Figure 5-61 The completed DFU beans example

## 5.6 Support for export, compile, run iSeries programs

This support allows you to export, compile, and run Java programs on an iSeries server from the VisualAge for Java IDE. It allows you to customize and save options.

### 5.6.1 Setup

Before using ET/400 support to export, compile, or run iSeries Java programs, select the **Properties** option from the ET/400 menu. Set up the options that you want to use. See Figure 5-62.

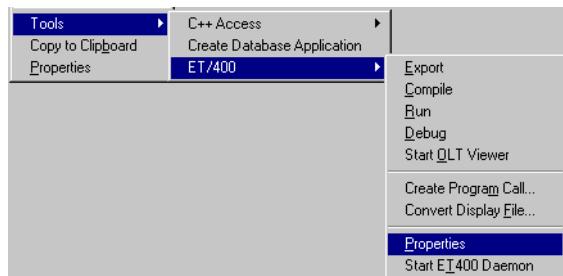


Figure 5-62 Setting ET/400 properties

The AS/400 Properties menu is shown in Figure 5-63 on page 294. Export Options allows you to control the name of the iSeries server to which you are exporting and the directory in the iSeries integrated file system to which you want to export.

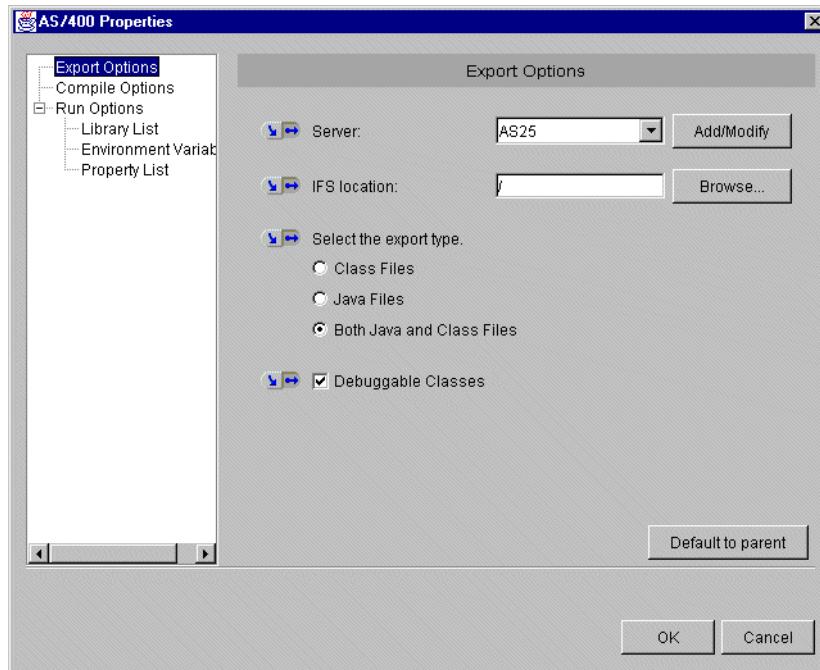


Figure 5-63 AS/400 Properties

Compile Options, as shown in Figure 5-64, allow you to set the optimization level, whether to replace existing programs or to enable the collection of performance information.

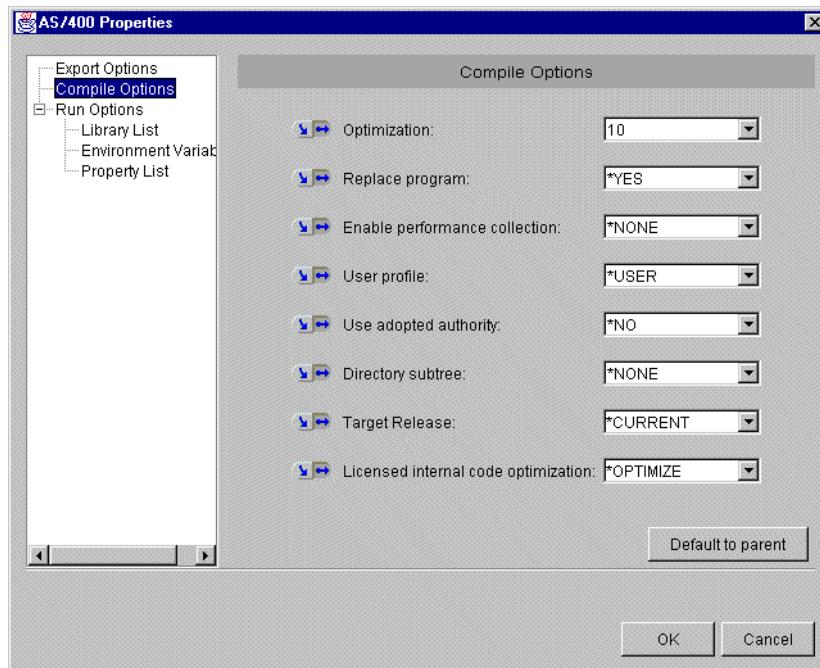


Figure 5-64 Compile Options

Run Options, as shown in Figure 5-65, allow you to set Parameter values, the Library List, Environment variables, and Property List.

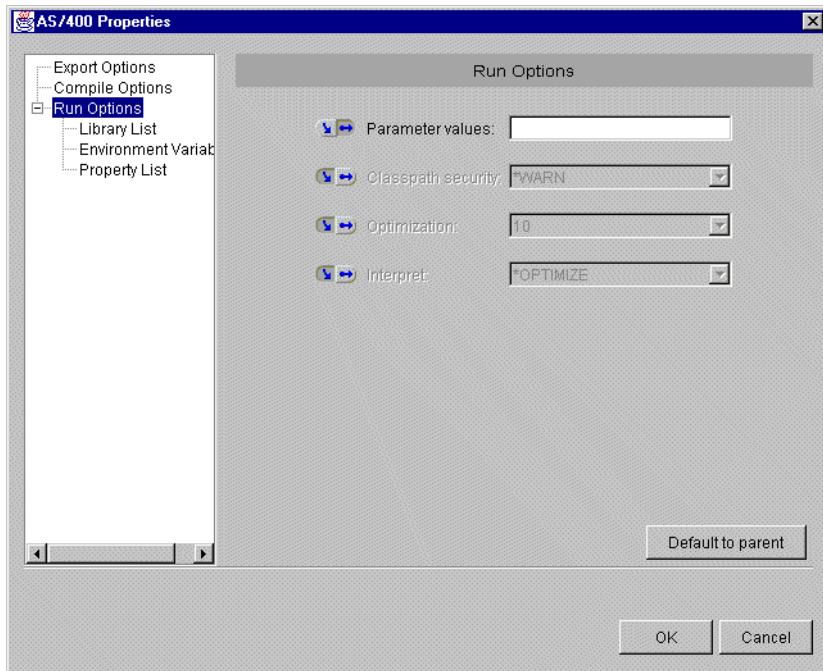


Figure 5-65 Run Options

## 5.6.2 Export support

Export support helps you export Java files from the VisualAge for Java Integrated Development Environment to the iSeries integrated file system (IFS). If you have a network drive assigned to the iSeries server, you can export directly from VisualAge for Java to the IFS. Using the ET/400 export support, you do not need a network drive. To use the ET/400 support, highlight the classes that you want to export, and select Export from the ET/400 menu. You can export multiple classes at once.

## 5.6.3 Compile support

To compile a program on the iSeries server, highlight the class file. You can have multiple classes highlighted. Select **Compile** from the ET/400 menu. The compile iSeries Java class files support creates an iSeries Java program from a Java class file. The resulting Java program object becomes part of the class file object, but cannot be viewed or modified directly. The Java class file name must be in one of the following iSeries integrated file systems:

- ▶ QOpenSys
- ▶ "root"
- ▶ A user-defined file system

Behind the scenes, this support calls the Create Java Program (CRTJVAPGM) command on the iSeries server and returns messages back to the user. The CRTJVAPGM process is done the first time you run a Java program on the iSeries server. If you have large programs that take a long time to compile, you may want to use this support so the program is created before you run it. The only extension of the CRTJVAPGM command provided is the ability to save the compile definitions locally, so you do not have to specify them the next time you compile the same class. For example, you may want to save the optimization level with which the iSeries Java program should be compiled.

The following optimization levels are valid:

- ▶ **10**: The Java program contains a compiled version of the class byte codes but has only minimal additional compiler optimization. Variables can be displayed and modified while debugging.
- ▶ **\*INTERPRET**: The Java program created is not optimized. When invoked, the Java program interprets the class file byte codes. Variables can be displayed and modified during debugging.
- ▶ **20**: The Java program contains a compiled version of the class file byte codes and has some additional compiler optimization. Variables can be displayed but not modified while debugging.
- ▶ **30**: The Java program contains a compiled version of the class file byte codes and has more compiler optimization than optimization level 20. During a debug session, user variables cannot be changed, but can be displayed. The presented values may not be the current values of the variables.
- ▶ **40**: The Java program contains a compiled version of the class byte file codes and has more compiler optimization than optimization level 30. All call and instruction tracing is disabled.

The output from the compile is returned to you in the VisualAge IDE in a dialog box as shown in Figure 5-66.

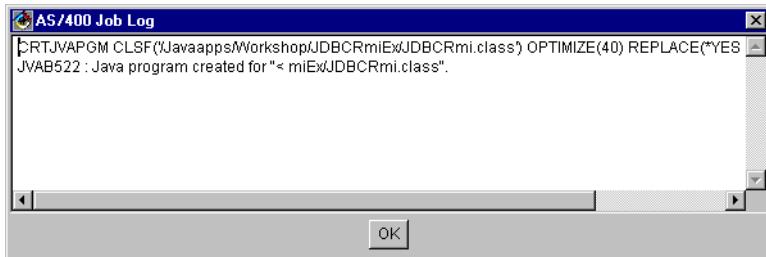


Figure 5-66 ET/400 compile dialog box

#### 5.6.4 Run support

ET/400 allows you to run an iSeries Java program from the Workbench and from browsers. Before you can run an iSeries Java program, you must complete these tasks:

1. Export the Java class files to an iSeries Integrated File System directory (Java source files are also required on the iSeries server to debug iSeries Java programs).
2. Set the run properties. If you want to run a Java program using Java 2 program, you need ensure that option 3 of 5769-JV1 is installed. You also need to specify a property(java.version) in the property list as shown in Figure 5-67, and click **Set**.

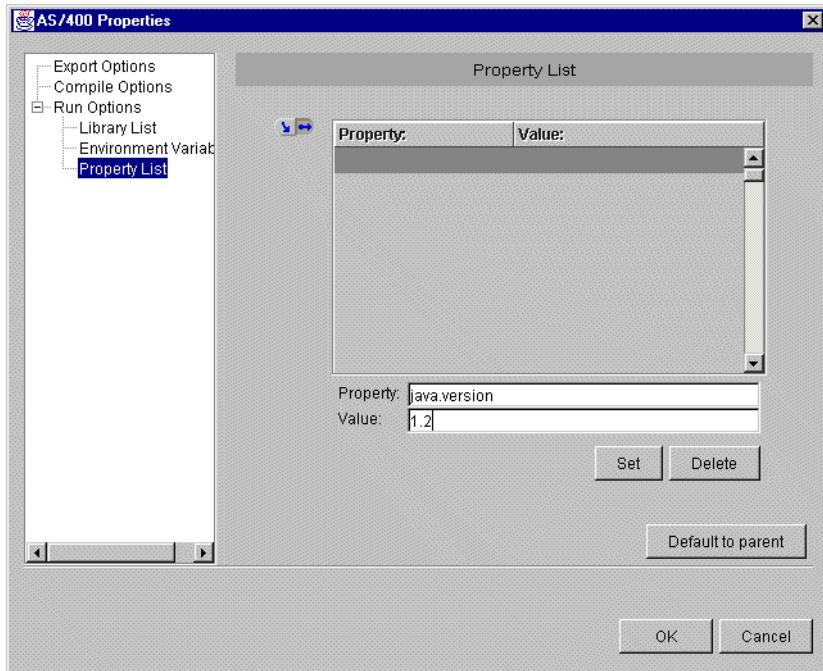


Figure 5-67 Set run properties

After you set the run options, you are ready to run the Java program.

1. Select a class, package, or project. All classes contained in the selected program element are run.
2. Select **Tools** from the pull-down menu. Select **ET/400** and then choose **Run**.

## 5.7 ET/400 system requirements

The ET/400 system requires a client platform with:

- ▶ VisualAge for Java Enterprise Edition or VisualAge for Java for iSeries
- ▶ A TCP/IP connection to the iSeries server

The iSeries server must be equipped with:

- ▶ Application Development Tool Set (ADTS): Required for the Display File Conversion SmartGuide. ADTS Version 3 Release 2 or Version 3 Release 6 and above is required.
- ▶ OS/400 V4R2 (or later) QJAVA library: Required to compile and run Java on the iSeries server.

iSeries connections must be established. By having ET/400 Daemon started on the workstation system, you can avoid signing onto the same iSeries server more than once. When the server is started, it keeps a connection to the iSeries active, so that it may be used by other ET/400 components.

To start the ET/400 Daemon, right-click in the Workbench window and select **Tools -> ET/400 -> Start ET400 Daemon** from the pop-up menu.





# Overview of the Order Entry application

This chapter covers the RPG Order Entry application example. This application is representative of a commercial application, although it does not include all of the necessary error handling that a business application requires.

This chapter introduces the application and specifies the database layout. In Chapter 8, “Migrating the user interface to the Java client” on page 321, we convert the RPG Order Entry application to a client/server application that uses Java to handle the data entry functions and RPG to handle the server database functions. The goal is to use the existing RPG application to service both the client application and the host 5250 application.

In Chapter 9, “Moving the server application to Java” on page 349, we convert the server-based RPG application to Java, so both sides of the application are written in Java.

## 6.1 Overview of the Order Entry application

This section provides an overview of the application and a description of how the application database is used.

### 6.1.1 The ABC Company

The ABC Company is a wholesale supplier with one warehouse and 10 sales districts. Each district serves 3,000 customers (30,000 total customers for the company). The warehouse maintains stock for the 100,000 items sold by the Company.

Figure 6-1 illustrates the company structure (warehouse, district, and customer).

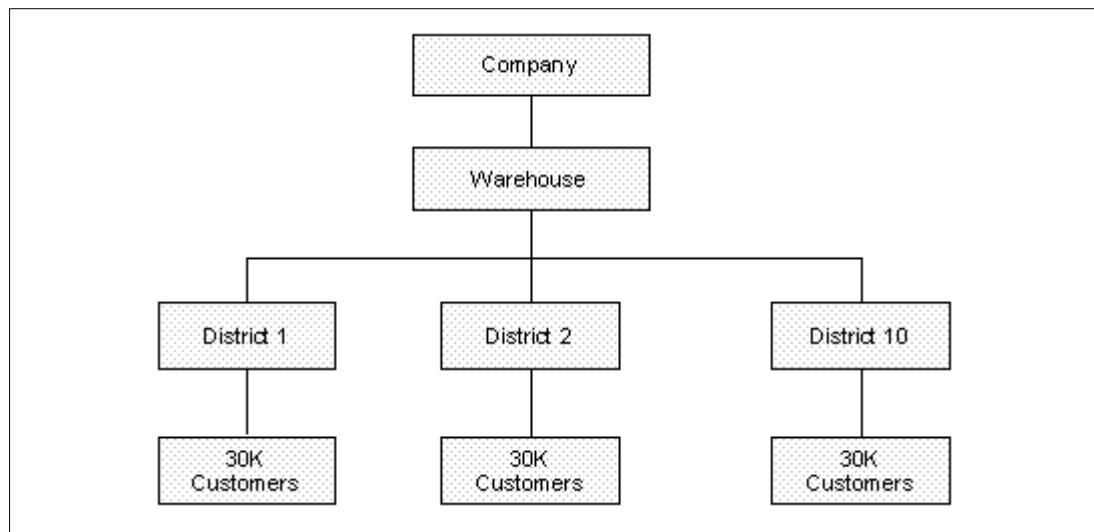


Figure 6-1 The company structure

### 6.1.2 The ABC Company database

The company runs its business with a database. This database is used in a mission critical, online transaction processing (OLTP) environment. The database includes tables with the following data:

- ▶ District information (next available order number, tax rate, and so on)
- ▶ Customer information (name, address, telephone number, and so on)
- ▶ Order information (date, time, shipper, and so on)
- ▶ Order line information (quantity, delivery date, and so on)
- ▶ Item information (name, price, item ID, and so on)
- ▶ Stock information (quantity in stock, warehouse ID, and so on)

### 6.1.3 A customer transaction

A customer transaction occurs based on the following series of events:

1. Customers telephone one of the 10 district centers to place an order.
2. The district customer service representative answers the telephone, obtains the following information, and enters it into the application:
  - Customer number
  - Item numbers of the items the customer wants to order
  - The quantity required for each item

3. The customer service representative may prompt for a list of customers or a list of parts.
4. The application then performs the following actions:
  - a. Reads the customer last name, customer discount rate, and customer credit status from the Customer Table (CSTMTR).
  - b. Reads the District Table for the next available district order number. The next available district order number increases by one and is updated.
  - c. Reads the item names, item prices, and item data for each item ordered by the customer from the Item Table (ITEM).
  - d. Checks if the quantity of ordered items is in stock by reading the quantity in the Stock Table (STOCK).
5. When the order is accepted, the following actions occur:
  - a. Inserts a new row into the Order Table to reflect the creation of the new order (ORDERS).
  - b. A new row is inserted into the Order Line Table to reflect each item in the order.
  - c. The quantity is reduced by the quantity ordered.
  - d. A message is written to a data queue to initiate order printing.

#### 6.1.4 Application flow

The RPG Order Entry Application consists of the following components:

- ▶ **ORDENTD (Parts Order Entry):** Display File
- ▶ **ORDENTR (Parts Order Entry):** Main RPG processing program
- ▶ **PRTORDERP (Parts Order Entry):** Print File
- ▶ **PRTORDERRR (Print Orders):** RPG server job
- ▶ **SLTCUSTD (Select Customer):** Display file
- ▶ **SLTCUSTR (Select Customer):** RPG SQL stored procedure
- ▶ **SLTPARTD (Select Part):** Display file
- ▶ **SLTPARTR (Select Part):** RPG stored procedure

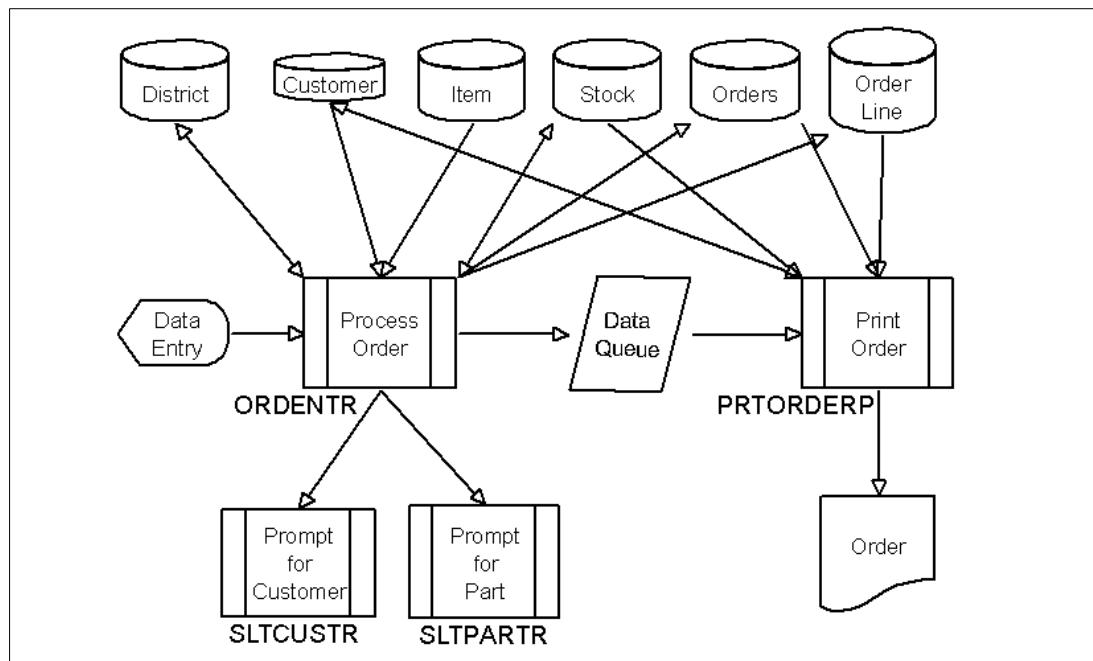


Figure 6-2 RPG application flow

ORDENTR is the main RPG program. It is responsible for the main line processing. It calls two supporting RPG programs that are used to prompt for and select end-user input. They are SLTCUSTR, which handles selecting a customer, and SLTPARTR, which handles selecting part numbers. PRTODERR is an RPG program that handles printing customer orders. It reads order records that were placed on a data queue and prints them in a background job.

### 6.1.5 Customer transaction flow

The following scenario walks through a customer transaction showing the application flow. By understanding the flow of the iSeries application, you can understand the changes made to this application to support a graphical client.

#### Starting the application

To start the application, the customer calls the main program from an iSeries command line:

```
CALL ORDENTR
```

When the Order Entry application is started, the display shown in Figure 6-3 appears.

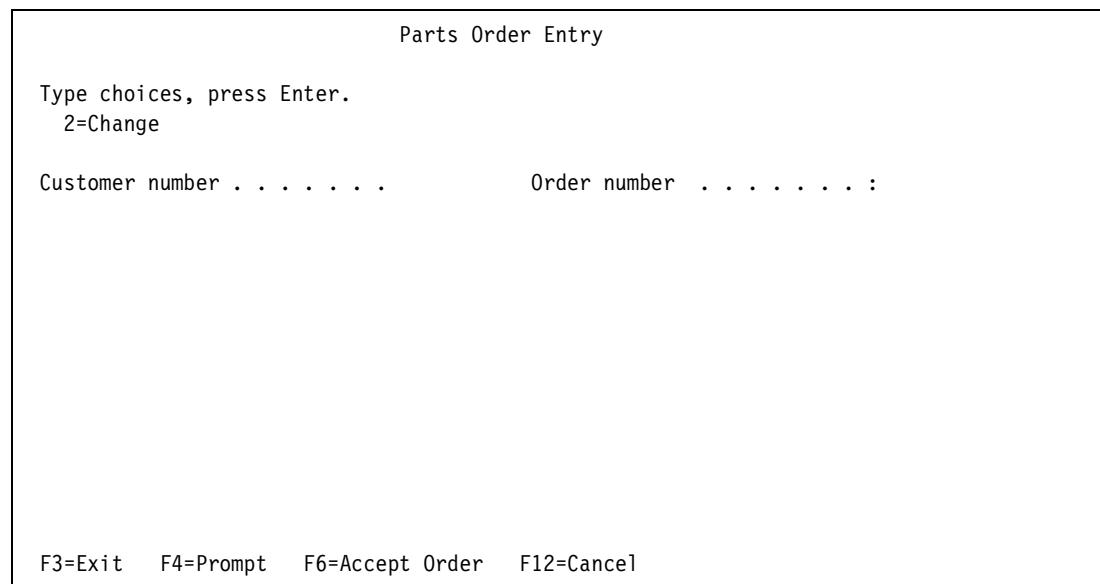


Figure 6-3 Parts Order Entry

When the Parts Order Entry display appears, the user has two options:

- ▶ Type a customer number and press the Enter key.
- ▶ End the program by pressing either F3 or F12.

If they do not know the customer number, the user can press F4 to view a window containing a list of available customers (Figure 6-4).

Select Customer

Type choices, press Enter.

1=Select

Opt Customer

- OAKLEY, Annie O
- BARBER, Elizabeth A
- Pork, Piggy B
- WILLIS, NEIL U
- Mullen-Schultz, Gary C
- MAATTA, Bob W
- FAIR, JIM J
- COULTER, SIMON S
- GOUDET, PIERRE W
- LLAMES, Joe L

More...

F12=Cancel

*Figure 6-4 Select Customer screen*

The user presses F12 to cancel the window and return to the initial panel, or they scroll through the items in the list until they find the customer they want. The user indicates their choice by typing 1 in the option field and pressing the Enter key. The selected customer is then returned to the initial panel (Figure 6-5).

Parts Order Entry

Type choices, press Enter.

2=Change

Customer number . . . . . : 0001      Order number . . . . . : 3548  
Customer name . . . . . : OAKLEY, Annie 0  
Address . . . . . . . . . : 00001 Ave. ABC  
                              Bldg 00001  
City . . . . . . . . . : Des\_Moines\_                  IO 07891-2345

Opt Part      Description                                   Qty

*Figure 6-5 Parts Order Entry*

After selecting a customer from the list, or typing a valid customer number and pressing the Enter key, the customer details are shown and an order number is assigned. An additional prompt is displayed that allows the user to type a part number and quantity.

If the user does not know the part number, they can press F4 to view a window containing a list of available parts (Figure 6-6 on page 304).

Select Part		
Type choices, press Enter.		
1>Select		
Opt	Part	Description
000001	WEBSPHERE REDBOOK	318
000002	Radio_Controlled_Plane	7
000003	Change_Machine	37
000004	Baseball_Tickets	899
000005	Twelve_Num_Two_Pencils	1,720
000006	Over_Under_Shotgun	1,310
000007	Feel_Good_Vitamins	37
000008	Cross_Country_Ski_Set	55
000009	Rubber_Baby_Buggy_Wheel	114
000010	ITSO REDBOOK SG24-2152	297
More...		
F12=Cancel		

*Figure 6-6 Select Part*

The user presses F12 to cancel the window and return to the initial panel, or they scroll through the items in the list until they find the part they want. The user indicates their choice by typing 1 in the option field and pressing the Enter key. The selected part is returned to the initial panel (Figure 6-7).

Parts Order Entry		
Type choices, press Enter.		
2=Change		
Customer number . . . . .	0001	Order number . . . . . : 3550
Customer name . . . . . :	OAKLEY, Annie 0	
Address . . . . . . . . . :	00001 Ave. ABC Bldg 00001	
City . . . . . . . . . :	Des_Moines_	I0 07891-2345
Opt	Part	Description
000008	Cross_Country_Ski_Set	Qty 2
F3=Exit F4=Prompt F6=Accept Order F12=Cancel		

*Figure 6-7 Parts Order Entry*

After selecting a customer from the list, or typing a valid customer number and pressing the Enter key, the part and quantity ordered are added to the list section below the part entry fields (Figure 6-8).

Parts Order Entry			
Type choices, press Enter.			
2=Change			
Customer number . . . . .	0001	Order number . . . . . :	3551
Customer name . . . . . :	OAKLEY, Annie O		
Address . . . . . . . . . :	00001 Ave. ABC Bldg 00001		
City . . . . . . . . . :	Des_Moines_	IO	07891-2345
Opt Part	Description	Qty	
2	000008 Cross_Country_Ski_Set	2	
	000001 WEBSPHERE REDBOOK	1	
Bottom			
F3=Exit F4=Prompt F6=Accept Order F12=Cancel			

*Figure 6-8 Part Order Entry*

The user may type a 2 beside an entry in the list to change the order. When the user presses the Enter key, a window appears that allows the order line to be changed (Figure 6-9).

Change Selected Order		
000008	Cross_Country_Ski_Set	1
F4=Prompt	F12=Cancel	

*Figure 6-9 Changing the order quantity*

The user chooses to press F12 to cancel the change, press F4 to list the parts, or type a new part identifier or different quantity. Pressing the Enter key validates the part identifier and quantity. If valid, the order line is changed in the list, and the window is closed. The complete order is shown in Figure 6-10 on page 306.

```

Parts Order Entry

Type choices, press Enter.
2=Change

Customer number . . . . . 0001      Order number . . . . . : 3551
Customer name . . . . . : OAKLEY, Annie O
Address . . . . . . . . . : 00001 Ave. ABC
                           Bldg 00001
City . . . . . . . . . : Des_Moines_          IO 07891-2345

Opt Part    Description           Qty

000008  Cross_Country_Ski_Set     1
000001  WEBSPHERE REDBOOK        1

Bottom
F3=Exit   F4=Prompt   F6=Accept Order   F12=Cancel

```

*Figure 6-10 Completed order*

In Figure 6-11, you see the quantity for Cross Country Ski Set is changed to 1. When the order is complete, the user presses F6 to update the database. Then, an order is placed on the data queue for printing.

```

Display Spooled File
File . . . . . : PRTORDERP          Page/Line  1/2
Control . . . . .                               Columns  1 - 78
Find . . . . .
*....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+...
                           ABC Company - Part Order
OAKLEY, Annie O                      Order Nbr:      3551
00001 Ave. ABC                      Order Date:  3-27-2001
Bldg 00001
Des_Moines_          IO 07891-2345
Part    Description           Quantity  Price   Discount  Amount
=====
000008  Cross_Country_Ski_Set     1       $ 93.00  .1140    $92.89
000001  WEBSPHERE REDBOOK        1       $ 30.00  .1140    $29.96
-----
Order total:                                $122.85
=====

Bottom
F3=Exit   F12=Cancel   F19=Left   F20=Right   F24=More keys

```

*Figure 6-11 Printed order*

The printed order (Figure 6-11) is created by a batch process. It shows the customer details and the items, quantities, and cost of the order.

### 6.1.6 Database table structure

The ABC Company database has eight tables:

- ▶ District
- ▶ Customer
- ▶ Order
- ▶ Order Line
- ▶ Item
- ▶ Stock
- ▶ Warehouse
- ▶ History

The relationships among these tables are shown in Figure 6-12.

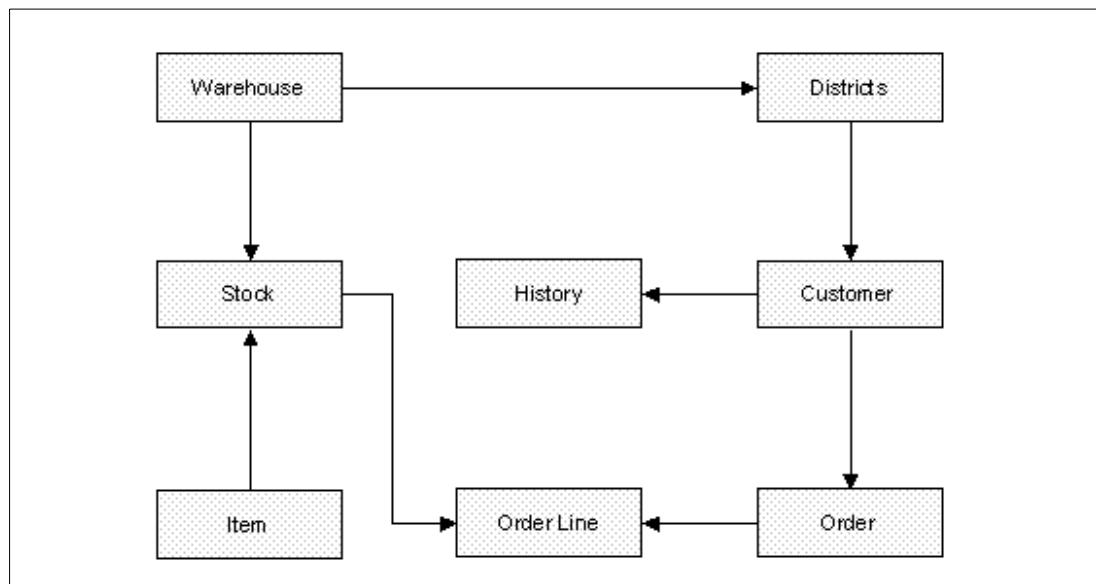


Figure 6-12 Table relationships

### 6.1.7 Order Entry application database layout

The sample application uses the following tables of the database:

- ▶ District
- ▶ Customer
- ▶ Order
- ▶ Order line
- ▶ Stock
- ▶ Item (catalog)

The following tables describe, in detail, the layout of the database.

## Tables

*Table 6-1 District Table Layout (Dstrct)*

Field name	Real name	Type	Length
DID	District ID	Decimal	3
DWID	Warehouse ID	Character	4
DNAME	District Name	Character	10
DADDR1	Address Line 1	Character	20
DADDR2	Address Line 2	Character	20
DCITY	City	Character	20
DSTATE	State	Character	2
DZIP	Zip Code	Character	10
DTAX	Tax	Decimal	5
DYTD	Year to Date Balance	Decimal	13
DNXTOR	Next Order Number	Decimal	9
Primary Key: DID and DWID			

*Table 6-2 Customer Table Layout (CSTMR)*

Field name	Real name	Type	Length
CID	Customer ID	Character	4
CDID	District ID	Decimal	3
CWID	Warehouse ID	Character	4
CFIRST	First Name	Character	16
CINIT	Middle Initials	Character	2
CLAST	Last Name	Character	16
CADDR1	Address Line 1	Character	20
CCREDIT	Credit Status	Character	2
CADDR2	Address Line 2	Character	20
CDCT	Discount	Decimal	5
CCITY	City	Character	20
CSTATE	State	Character	2
CZIP	Zip Code	Character	10
CPHONE	Phone Number	Character	16
CBAL	Balance	Decimal	7
CCRDLIM	Credit Limit	Decimal	7
CYTD	Year to Date	Decimal	13

Field name	Real name	Type	Length
CPAYCNT	Payment	Decimal	5
CDELCNT	Delivery Qty	Decimal	5
CLTIME	Time of Last Order	Numeric	6
CDATA	Customer Information	Character	500
Primary Key: CID, CDID, and CWID			

Table 6-3 Order Table Layout (ORDERS)

Field name	Real name	Type	Length
OWID	Warehouse ID	Character	4
ODID	District ID	Decimal	3
OCID	Customer ID	Character	4
OID	Order ID	Decimal	9
OENTDT	Order Date	Numeric	8
OENTTM	Order Time	Numeric	6
OCARID	Carrier Number	Character	2
OLINES	Number of Order Lines	Decimal	3
OLOCAL	Local	Decimal	1
Primary Key: OWID, ODID, and OID			

Table 6-4 Order Line Table Layout (ORDLIN)

Field name	Real name	Type	Length
OID	Order ID	Decimal	9
ODID	District ID	Decimal	3
OWID	Warehouse ID	Character	4
OLNBR	Order Line Number	Decimal	3
OLSPWH	Supply Warehouse	Character	4
OLIID	Item ID	Character	6
OLQTY	Quantity Ordered	Numeric	3
OLAMNT	Amount	Numeric	7
OLDLVD	Delivery Date	Numeric	6
OLDSTI	District Information	Character	24
Primary Key: OLWID, OLDID, OLOID, and OLNBR			

*Table 6-5 Item Table Layout (ITEM)*

Field name	Real name	Type	Length
IID	Item ID	Character	6
INAME	Item Name	Character	24
IPRICE	Price	Decimal	5
IDATA	Item Information	Character	50
Primary Key: IID			

*Table 6-6 Stock Table Layout (Stock)*

Field name	Real name	Type	Length
STWID	Warehouse ID	Character	4
STIID	Item ID	Character	6
STQTY	Quantity in Stock	Decimal	5
STD101	District Information	Character	24
STD102	District Information	Character	24
STD103	District Information	Character	24
STD104	District Information	Character	24
STD105	District Information	Character	24
STD106	District Information	Character	24
STD107	District Information	Character	24
STD108	District Information	Character	24
STD109	District Information	Character	24
STD110	District Information	Character	24
STYTD	Year to Date	Decimal	9
STORDERS	Number of orders	Decimal	5
STREMORD	Number of remote orders	Decimal	5
STDATA	Item Information	Character	50
Primary Key: STWID and STIID			

## 6.1.8 Database terminology

This Redbook concentrates on the use of the iSeries server as a database server in a client/server environment. In some cases, we use SQL to access the iSeries database. In other cases, we use native database access.

The terminology used for the database access is different in both cases. In Table 6-7, you find the correspondence between the different terms.

*Table 6-7 Database terminology*

iSeries native	SQL
Library	Collection, Schema
Physical File	Table
Field	Column
Record	Row
Logical File	View or Index





# Java for RPG programmers

This chapter provides a brief overview of Java and object-oriented programming from the perspective of an RPG programmer. If you are already familiar with OO, Java, or C++, you may not need to read this information.

Java is an object-oriented (OO) programming language. RPG is not. However, many of the concepts from OO can be translated and applied to RPG. In this chapter, the similarities between OO languages and high-level languages (HLL) are discussed. There is also an introduction to the Java language. You can obtain a complete introduction to Java from any of the current books on the subject that are available at your local book store.

This chapter contains information about these subjects:

- ▶ Fields as variables
- ▶ Procedures/subroutines as methods
- ▶ Modules as classes
- ▶ Programs as packages
- ▶ The Java language

## 7.1 Object-oriented programming and RPG

No, this is not an oxymoron. It is quite possible to write OO programs in any programming language. It merely requires a little discipline on the part of the programmer. This section discusses how an application can implement OO principles in RPG as a bridge to the new terminology that is associated with OO programming.

RPG IV is a closer fit to OO than the earlier versions of RPG, so we confine our comparison to that version.

### **Variables**

Variables equate to RPG fields. There are many different kinds of variables:

- ▶ **Class variables:** Class variables are visible to all instances of a class. They are declared with the *static* keyword. They are shared among those objects, so that changing the variable in one object affects all the objects.
- ▶ **Instance variables:** Instance variables are scoped to a specific object. They may be global to the object or scoped to a method or block.
- ▶ **Local variables:** Local variables are instance variables scoped to a particular method or block. The variable is not visible outside of the method or block and, therefore, cannot be accessed by other methods or blocks.
- ▶ **Final variables:** Final variables are used to define constants. They are usually assigned a value when they are declared, but may be assigned a value once only while a method is running if a value has not yet been assigned.

RPG IV implements these constructs as:

- ▶ **Global fields:** Global fields are the normal form of field in RPG. These are declared on the *D-spec* of the main program. These fields are visible to the entire program and may be referenced and modified from anywhere in the program.
- ▶ **Local fields:** Local fields are scoped to a procedure. They are declared on the *D-spec* of the procedure and cannot be modified by other procedures.
- ▶ **Named constants:** Named constants are used to provide meaningful names for *magic* values in a program. Rather than coding -1, -2, -4, -8, and so on as indicators of error severity, the negative numbers are assigned names so the program can reference words such as DIAG, WARN, ERROR, FATAL, and so on.

### **Methods**

Methods equate to RPG IV procedures or subroutines. Procedures are a closer fit because they provide better support for encapsulation through interface prototyping and local fields. Methods and procedures are where the real work is performed. They contain the code that actually performs the function. For example, a method or procedure may provide support for converting a date from Year/Month/Day format to Day/Month/Year format.

### **Classes**

Classes equate to RPG IV modules. A class is a collection of methods and variables. A module is a collection of procedures or subroutines. Classes and modules are designed to support a particular function. For example, a series of date conversion methods or routines may be grouped in a single class or module.

### **Packages**

Packages equate to RPG IV programs or service programs. They are a means of grouping similar functions together. A package contains one or more classes. A program or service program contains one or more modules. For example, a series of conversion classes or modules may be grouped in a single package or service program.

### **Differences between Java and RPG**

Java supports a number of modifiers when declaring classes and variables that do not have direct equivalents in RPG. Some of these modifiers include:

- ▶ **Public:** The variable or method is visible to the users of the class.
- ▶ **Private:** The variable or method is not visible to the users of the class. Only the class can use it.
- ▶ **Protected:** The variable or method is visible to the class in which it is defined and also to the subclasses of that class.

You can implement the public and private modifiers in RPG IV by using the features of the Integrated Language Environment (ILE). A module may choose to export the fields and procedures it defines. Exported fields and procedures are available to the caller of the module (therefore, public). All other fields and procedures are private.

#### **7.1.1 What is Java**

Java originated as a language to program electronic consumer devices, such as microwave ovens, washing machines, and toasters. Software for these devices needs to be particularly reliable and must work on a variety of computer chips.

A small group of people were working on this problem at Sun Microsystems, Inc. and realized that languages, such as C and C++ were not suitable for this task. The reason is because C and its derivatives require a compiler that is specific to the computer chip that is being used in the device and the nature of C makes it difficult to write reliable software. The Sun group started developing a language to solve these problems.

These developers soon realized that the new language was ideally suited to the Internet due to the platform-independent nature of its architecture. This allowed a program to run on any systems that provided support for the runtime.

More information about the development of Java can be found in *The Java White Papers* at: <http://java.sun.com/docs/white/index.html>

Java is quite a simple language in that the basic constructs are few. However, similar to all OO languages, the complexity is in understanding the classes and methods.

Java is designed to support distributed applications through classes for Distributed Program Call, Remote Method Invocation (RMI), JDBC, and sockets. Java simplifies writing distributed applications by hiding much of the communications effort. Applications can be written to open files over the Internet simply by providing a Uniform Resource Locator (URL). By using the sockets classes, you can easily write client/server applications.

Java is intended to be used in a graphical environment. Graphical user interfaces (GUIs) tend to allow many things to happen at once. Threads are a convenient way of allowing a process to handle many tasks simultaneously. Writing code in C or C++, which deals with threads, is a particularly onerous task. Java simplifies writing a threaded application by providing built-in language support for threads.

Java code is intended to be portable. Because Java bytecodes run in a Java virtual machine (JVM), the compiled program is platform-independent. It can run on any hardware that implements a JVM. Bytecodes are generally interpreted by the JVM, although some platforms provide a compiler to improve performance. Writing an application in Java can increase the usefulness of the application by removing hardware restrictions.

While Java is an OO language, it is not a pure OO language, because it makes a distinction between various types of data. Most things are objects, but Java supports so-called primitive data types. These primitive data types are the basic kinds of program data:

- ▶ **boolean**: A 1-bit value representing true or false
- ▶ **char**: A 16-bit value representing a single Unicode character
- ▶ **byte**: An 8-bit value representing a signed integer
- ▶ **short**: A 16-bit value representing a signed integer
- ▶ **int**: A 32-bit value representing a signed integer
- ▶ **long**: A 64-bit value representing a signed integer
- ▶ **float**: A 32-bit value representing an IEEE 754 floating point number
- ▶ **double**: A 64-bit value representing a floating point number

All other data types are objects and derive from the object class. They are generally more complex data types.

As a contrast, Smalltalk implements everything, including primitive types, as a subclass of Object.

### 7.1.2 Java syntax

The Java syntax is similar to the C language. This is because Java was partly designed as a replacement for C (a better C) and also to appeal to the huge number of existing C and C++ programmers.

Each line of Java source may span multiple physical lines in the source file. Each logical line ends with a semicolon.

**Note:** Lines that use braces are not ended with semicolons. The statements may be entered in any format and blank lines are ignored.

Statements are grouped using braces. Braces delimit scope blocks, the beginning and end of methods, and the beginning and end of classes.

```
if (someCondition)
{
    // do this stuff
}
else
{
    // do this other stuff
}
```

All names in Java are case-sensitive, for example:

- ▶ int myInteger;
- ▶ int MyInteger;
- ▶ int myinteger;
- ▶ int MYINTEGER;

These four integer variables are all different entities in Java. The RPG compiler folds these names to uppercase and treats them as a single entity.

Comments may begin with either the C style of a slash followed by an asterisk and end with an asterisk followed by a slash, or the C++ style of a double slash:

```
/* The first line of a C style comment line
** The second line of a C style comment line
** The final line follows ....
*/
// A C++ style comment line
// Another C++ style comment line
```

Java is also a strongly typed language. This means that all named entities in the program must have a specific type (for example, char, int, Object, and so on). The type is always specified when declaring a variable in Java.

### 7.1.3 Object creation

Every class in Java has at least one constructor method. This is a method with the same name as the class. It may or may not accept arguments. An object is created by creating a new instance of a class:

```
Thing myThing = new Thing();
```

This statement performs both the definition and the declaration of a variable. The definition is the part to the left of the equal sign and says define a variable called myThing that is a type of Thing. The declaration is the part to the right of the equal sign and says create a new Thing. The equal sign is an assignment statement. The statement creates a new Thing and stores the reference to that new object in the variable myThing.

Thing() is the default constructor for the Thing class and does not accept any arguments. It is possible for a class to have multiple constructors each accepting different arguments. A constructor is used to initialize a new object.

### 7.1.4 Class variables

Class variables are those with a static modifier. These variables are associated with the class and, therefore, are accessible from every instance of the class. They are used where the variable represents something that is either independent of each object or is dependent on all objects. For example, a counter of the number of objects (instances of a particular class) can be implemented as a class variable and increased by the constructor for that class giving all objects knowledge of how many of them exist:

```
static int howMany = 0;
```

### 7.1.5 Class methods

Class methods also use the static modifier. You can start these methods without an instance of the class having been created. This may be necessary where the methods operate on one or more of the various primitive data types. For example, the Math class (in java.lang) declares all its methods as static, because no object is required to use the Math functions. They all accept numeric primitive data types. They also do not reference any object instance data.

```
double aRoot = Math.sqrt(someValue);
```

## 7.1.6 Instance variables

Instance variables are specific to an object and are not shared among other objects. They may be visible to other objects (in which case, they are said to be *public*) or hidden from other objects (*private* – most instance variables are private). They can be public, private, protected, or default.

## 7.1.7 Instance methods

Instance methods are only usable when an object has been created. They provide the code that implements the programming logic for the class. Instance methods may be public, private, protected, or default.

## 7.1.8 Thread

Thread is a Java way to make a single Java virtual machine look like many machines and all running at the same time. Java provides a number of tools for creating and managing threads. This facility helps us to create the objects which can be running concurrently.

## 7.1.9 Object destruction

Objects are not explicitly destroyed in Java. They are automatically removed by the garbage collector when there are no further references to the object. This work is usually done in the background by a low priority thread. The idea of a garbage collector has existed for a long time and has been implemented in languages such as Smalltalk and Lisp. The JVM knows which objects it has allocated. It can also determine which variables reference which objects. It has algorithms that can determine when an object is no longer needed.

There is an important consequence of the nature of automatic garbage collection. The objects cannot be collected if we allow accessible references to those unnecessary objects in our program. Therefore, it may be a good idea to explicitly assign **null** value into a variable when it is no longer needed:

```
myThing = null;
```

## 7.1.10 Subclasses and inheritance

Inheritance is one of the more powerful features of an OO language. It allows code to be reused by creating a subclass of an existing class. The new class gets all of the code in the parent class (super class) and can extend the parent class by providing its own routines to do things that the parent class was not designed to do.

## 7.1.11 Overloading and overriding methods

There are some circumstances when we want to re-use a method of its super class. There are two ways that we can do this with Java. Reusing the same method name with different arguments and perhaps a different return type is known as *overloading*. Using the same method name with identical arguments and return type is known as *overriding*.

This allows a class to change the behavior of a method that is provided by its super class. For example, a class hierarchy that represents various geometric shapes can have an Ellipse class that inherits from a Circle class. Both classes may need a means of returning the size of their area. However, calculating the area of an ellipse is different from calculating the area of a circle so the Ellipse class can override the Circle area() method.

## 7.1.12 Compiling Java on the iSeries server

Java programs on the iSeries server are stored in the integrated file system. They run as Java bytecodes or direct execution programs.

Running a Java program from bytecodes results in exactly the same form of execution as all other JVMs. However, the iSeries server supports a direct execution mechanism where the Java bytecodes for a class are transformed into a service program that results in faster processing.

You can compile Java on the iSeries server by using any of these commands:

- ▶ Use the **javac** command within the Qshell Interpreter on the iSeries server or within the SDK on a PC that compiles the Java source into bytecodes. The resulting .class file is stored in the integrated file system.
- ▶ Run the Create Java Program (**CRTJVAPGM**) command from an iSeries command line to create a direct execution version of the .class file.
- ▶ Run the **java** command or Run Java (**RUNJAVA**) command, which automatically creates a direct execution version of the Java program.

For a more detailed coverage of Java and RPG, see the *Java for RPG Programmers* book by Phillip Coulthard and George Farr.





## Migrating the user interface to the Java client

This chapter covers the steps that are necessary to create a Java Graphical User Interface (GUI) that interacts with the Order Entry application that was discussed in Chapter 6, “Overview of the Order Entry application” on page 299. The user interface is designed, so minimal changes are needed in the RPG code. Furthermore, the host application is changed, so it can handle both an invocation from a Java client and a native invocation that does not involve a Java interface.

## 8.1 Re-designing the application

Figure 8-1 shows the original design of the Order Entry application.

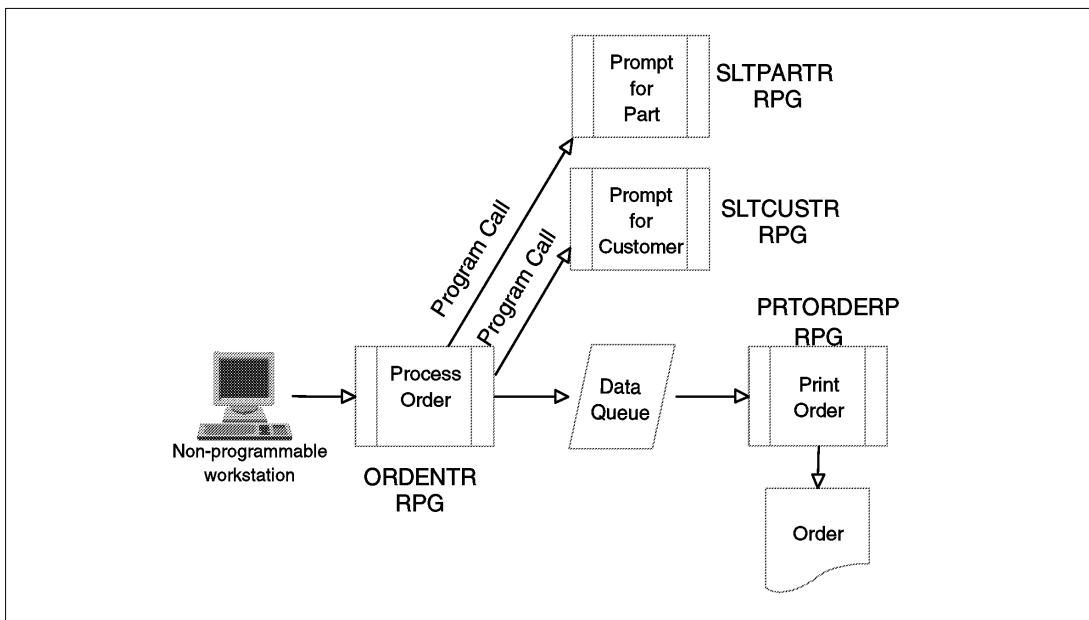


Figure 8-1 Original Order Entry application

We migrate the application to a Java client that provides a GUI. We modify the original RPG code to allow it to be used from either the new Java client interface or from the original 5250 interface. Figure 8-2 shows the new design.

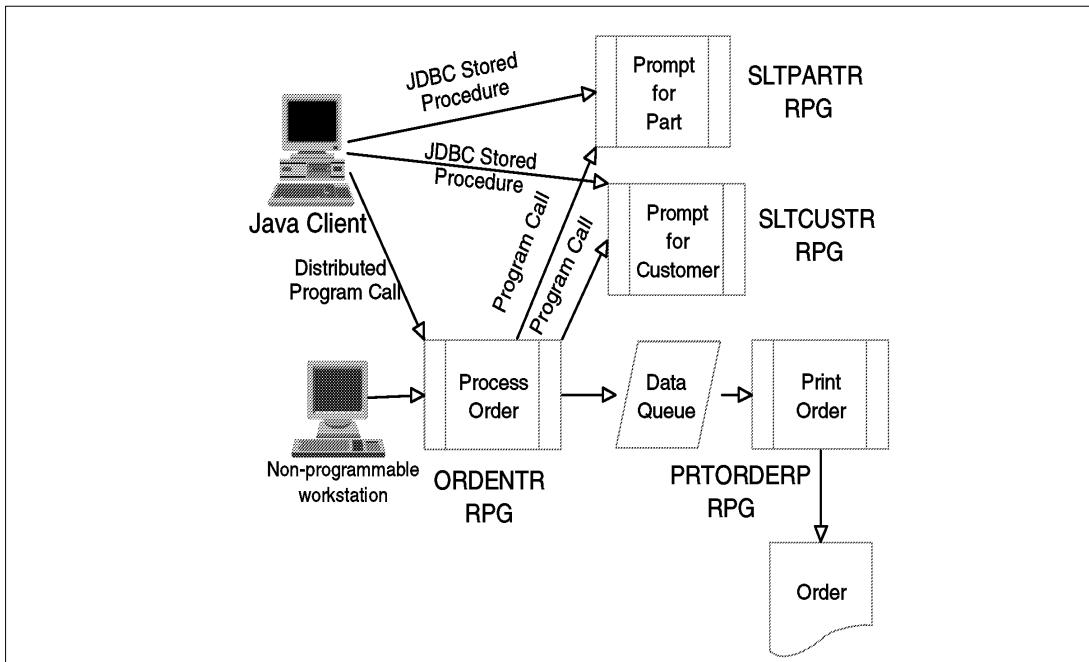


Figure 8-2 Java client Order Entry application

## 8.2 Creating the Java client graphical user interface

First, we analyze the steps and code that are involved in creating the Java GUI. It is built using VisualAge for Java Version 3.5. The IBM Toolbox for Java classes are used to access data and programs on the iSeries server. These IBM Toolbox for Java topics are covered:

- ▶ Stored procedures using the IBM Toolbox for Java JDBC driver
- ▶ DDM record-level access
- ▶ Distributed Program Call

We assume that you are familiar with IBM VisualAge for Java and have a basic understanding of the IBM Toolbox for Java. For more information about these topics, see Chapter 3, “Introduction to VisualAge for Java” on page 63, and Chapter 4, “IBM Toolbox for Java” on page 123.

We also discuss the changes that are made in the host RPG application later in this chapter. First, we focus on the “look” of the window that is designed to interact with the Order Entry application. We discuss the window components. Subsequent sections explore the issues that relate to the functionality, which is associated with the individual components in the window.

## 8.3 Overview of the Parts Order Entry window

Figure 8-3 shows the main Order Entry window. This window is built using VisualAge for Java. The name of the class that defines this window is OrderEntryWdw. It is the controlling class, or entry point, of the client application.

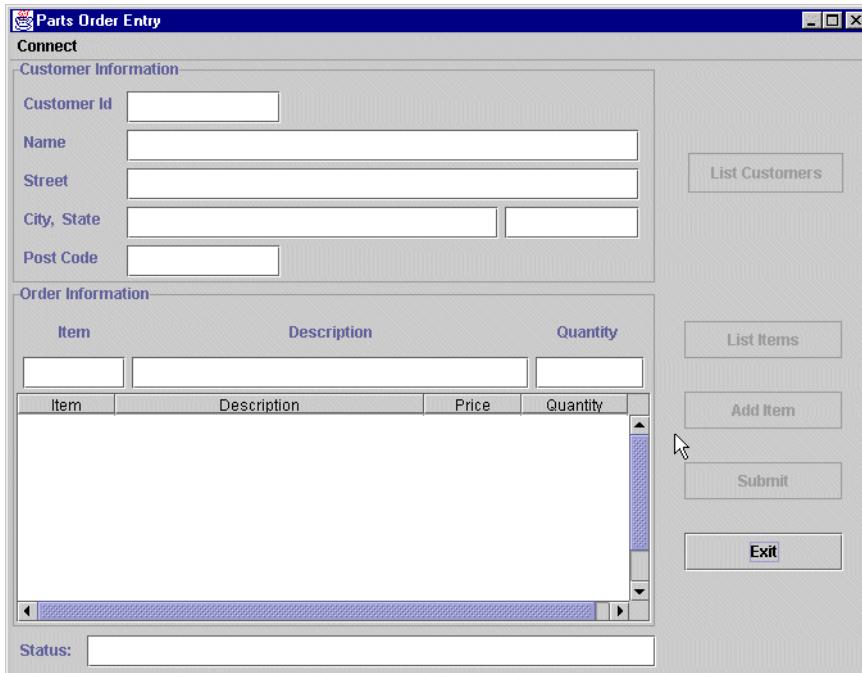


Figure 8-3 Parts Order Entry: Initial panel

The primary window components are:

- ▶ A menu bar that contains a Connect menu item
- ▶ Six text fields for the customer information
- ▶ Three text fields for the current item that is selected

- ▶ A table that displays all items in the current order
- ▶ A text field for status updates
- ▶ A button for listing all valid customers
- ▶ A button for listing all valid items that can be ordered
- ▶ A button for adding the current item to the order list box
- ▶ A button for submitting the order
- ▶ A button for exiting the application

When the window is first displayed, all of the buttons, except the Exit button, are disabled. Initially, the only valid options are exiting or connecting to the remote database (the host iSeries server).

This Java code is a partial listing of the class definition for OrderEntryWdw:

**Program listings:** For complete listings of all the code examples shown throughout this book, refer to Appendix A, “Additional material” on page 529. This appendix includes instructions for accessing the Redbooks Web site and downloading the example code.

```
import java.sql.*;
import java.math.*;
import com.ibm.as400.access.*;
import orderObjects.*;
public class OrderEntryWdw extends javax.swing.JFrame {
    private String password = null;
    private String systemName = null;
    private String userid = null;
    private AS400 as400 = null;
    private Connection dbConnect = null;
    private KeyedFile itemFile = null;
    // not shown are all the TextFields, Buttons, etc.
    // generated by VisualAge for Java
}
```

We only show the data members that are not added by the VisualAge for Java Composition Editor. There are three string objects for sign-on information. In addition, there is an SQL Connection object (dbConnect) that is created from the connection class, which is included with the java.sql package. Finally, we declare two objects from the classes that the IBM Toolbox for Java provides: an AS400 object and a KeyedFile object. Now, we examine how these objects (in conjunction with GUI controls and other objects) interface with the RPG Order Entry application.

## 8.4 Application flow through the Java client Order Entry window

The client application supports this series of tasks:

- ▶ Connect to the remote database.
- ▶ Retrieve a list of valid customers.
- ▶ Select a customer.
- ▶ Retrieve a list of valid items (parts).
- ▶ Select an item.
- ▶ Verify the item.
- ▶ Add the item to the order.
- ▶ Submit the order.

We examine each of these tasks in the following sections.

### 8.4.1 Connecting to the database

Once the Parts Order Entry window has initially displayed, choose the **Connect** menu option to connect to the iSeries server. See Figure 8-4.

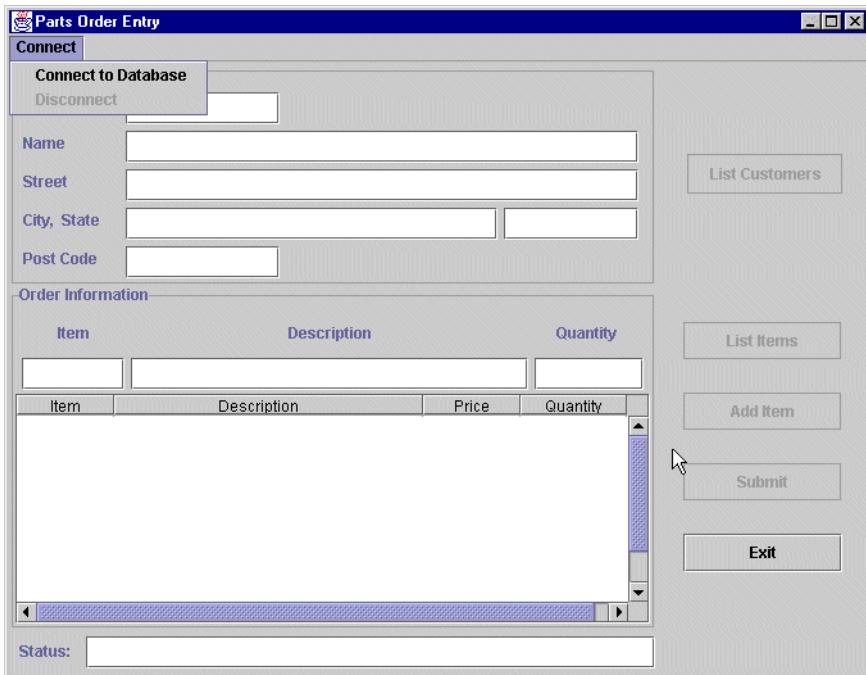


Figure 8-4 Parts Order Entry: Database Connect

A dialog is shown that requests sign-on information. You must enter a system name, user ID, and password, as shown in Figure 8-5. Port will be used when we explain RMI in Chapter 9, "Moving the server application to Java" on page 349.

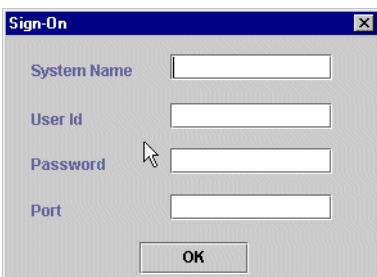


Figure 8-5 Sign-On to the system

Click the **OK** button. Then a connection is created to the `connectToDb()` method of the `OrderEntryWdw` class.

Here is the code for this method:

```
private boolean connectToDB(String systemName, String userid, String password)
{
    // This method invokes the openItemFile() method and then
    // establishes the JDBC connection

    updateStatus("Connecting to " + systemName + " ...");
```

```

        this.systemName = systemName;
        this.userid = userid;
        this.password = password;

        openItemFile();

        try
        {
            DriverManager.registerDriver(new com.ibm.as400.access.AS400JDBCDriver());
            dbConnect = DriverManager.getConnection("jdbc:as400://" + systemName +
                "/apilib;naming=sql;errors=full;date format=iso;" +
                "extended dynamic=true;package=JavaMig;package library=apilib",
                userid, password);
        }
        catch (Exception e)
        {
            updateStatus("Connect failed");
            handleException(e);
            return false;
        }

        updateStatus("Connected to " + systemName);
        return true;
    }
}

```

This method has two main responsibilities. It executes the `openItemFile()` method and establishes a JDBC connection. The `openItemFile()` method opens an iSeries remote file using record-level access functionality that is provided by the IBM Toolbox for Java. This is discussed in greater detail later.

After running the `openItemFile()` method, the Toolbox JDBC driver is loaded with this statement:

```
DriverManager.registerDriver
    (new com.ibm.as400.access.AS400JDBCDriver());
```

Next, we establish the SQL connection by running the `getConnection()` method, which is a static method in the `DriverManager` class:

```
dbConnect = DriverManager.getConnection("jdbc:as400://" + systemName +
    "/apilib;naming=sql;errors=full;date format=iso;" +
    "extended dynamic=true;package=JavaMig;package library=apilib",
    userid, password);
```

A URL is passed to the `getConnection()` method. The `systemName` value is retrieved from a text field in the Sign-On dialog, as are the `userid` and `password` values that are passed in. The URL string also specifies a default library of `apilib`, standard SQL naming convention, full error message information, and ISO format for date fields. Extended dynamic support is also enabled. This allows us to store the SQL statements in packages on the iSeries server. This provides better performance than using dynamic SQL. The name of the package we store the statements in is the `JavaMig` in `apilib` library. The `updateStatus()` method simply updates the text in the status text field.

As previously mentioned, the `openItemFile()` method handles opening the ITEM file on the iSeries server. This is the code for the method:

```
public void openItemFile()
{
    // declare a path name for the itemFile
    QSYSObjectPathName fileName = new QSYSObjectPathName
```

```

        ("APILIB","ITEM","*FILE","MBR");
// initialize the itemFile so that it becomes a handle
// to the ITEM file on the iSeries
itemFile = new KeyedFile(getAs400(), fileName.getPath());

try
{
    getAs400().connectService(AS400.RECORDACCESS);
}
catch(Exception e)
{
    updateStatus("Error establishing RECORDACCESS connection");
    handleException(e);
    return;
}

RecordFormat itemFormat = null;

// retrieve the record format of the file. Some files
// may have multiple formats, in this case there is only one
try
{
    AS400FileRecordDescription recordDescription =
    new AS400FileRecordDescription (getAs400(),"/QSYS.LIB/APILIB.LIB/ITEM.FILE");
    itemFormat = recordDescription.retrieveRecordFormat()[0];
    itemFormat.addKeyFieldDescription("IID");
}
catch(Exception e)
{
    updateStatus("Error retrieving file format on ITEM file");
    handleException(e);
    return;
}

// set the record format and open options for the file
try
{
    itemFile.setRecordFormat(itemFormat);
    itemFile.open(AS400File.READ_ONLY,1,
                 AS400File.COMMIT_LOCK_LEVEL_NONE);
}
catch(Exception e)
{
    updateStatus("Error opening ITEM file");
    handleException(e);
    return;
}

updateStatus("Item File successfully opened...");

return;
}

```

The `openItemFile()` method gets the `as400` object from the accessor `getAs400()` method and establishes the `RECORDACCESS` connection for this object. The `itemFile` object is initialized, so it becomes a handle to the `ITEM` file on the `iSeries` server. After instantiating a `RecordFormat` object for this file, it is opened in read-only mode with a blocking factor of one. Since we are only reading the file, we do not use commitment control. We use a blocking factor of one, because we only need one record at a time. We use this file later in the application when we need to verify items that are being ordered.

This completes the discussion of connecting to the iSeries server. In summary, an SQL connection is established, and a handle to the ITEM file on the iSeries server is initialized. The ITEM file is then opened. The List Customers button is enabled, and we are ready to retrieve a list of valid customers from the iSeries Customer Master database.

### 8.4.2 Program interfaces

The Java client program uses a number of different interfaces, as shown in Figure 8-6, to access the iSeries server. All of these interfaces are provided by the IBM Toolbox for Java. JDBC stored procedures populate the customer table and the item table. DDM record-level access verifies the items that are ordered. The Distributed Program Call interface submits the order. In the following sections, we examine the coding implementation of these interfaces in more detail.

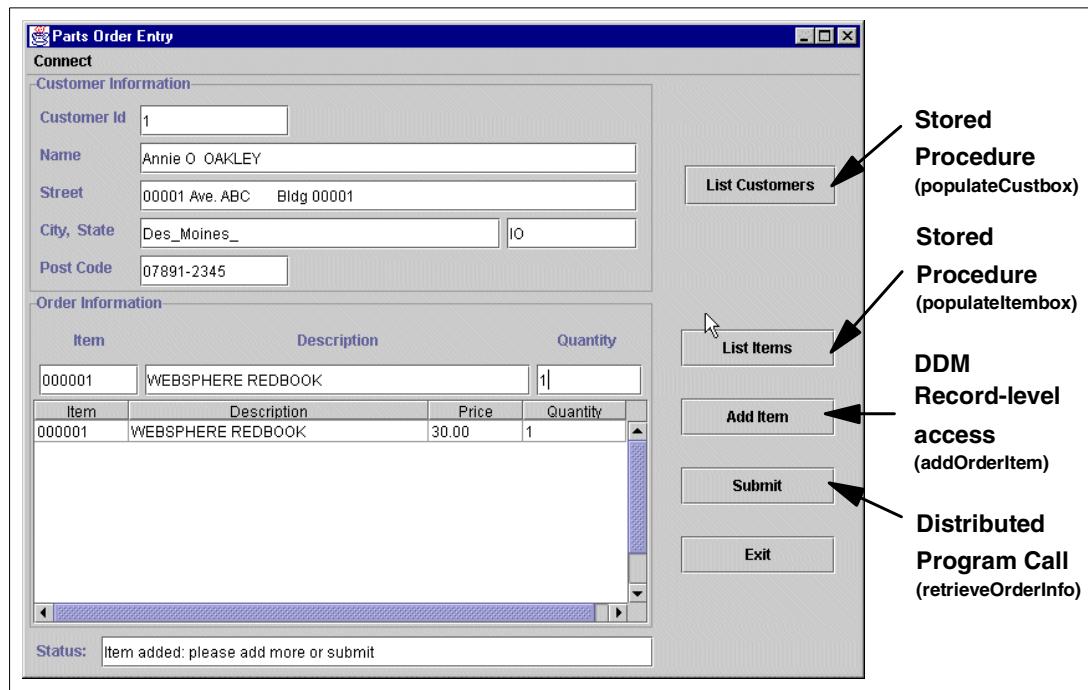


Figure 8-6 Java client programming interfaces

### 8.4.3 Retrieving the customer list

This section looks at the code that allows us to retrieve a list of valid customers from the iSeries server. Figure 8-7 shows the window that is displayed once the list of customers is retrieved.



Figure 8-7 Select Customer

SLtCustWdw is the class that defines this window. It contains a JTable object and one button. The JDBC interface is used to access an SQL stored procedure on the iSeries server. This stored procedure returns a result set. Records are fetched from the result set and the retrieved data populates the list box.

This code snippet shows the class definition for SLtCustWdw:

```
import java.sql.*;
import orderObjects.*;

public class SLtCustWdw extends javax.swing.JFrame implements
    java.awt.event.ActionListener, java.awt.event.WindowListener
{
    private Connection dbConnect = null;
    private OrderEntryWdw orderWindow = null;

    private javax.swing.JPanel ivjPanel1 = null;
    private javax.swing.JScrollPane ivjScrollPane1 = null;
    private javax.swing.JTable ivjScrollPaneTable = null;
    private javax.swing.table.DefaultTableModel ivjDefaultTableModel1 = null;
    private javax.swing.JPanel ivjButtonPanel = null;
    private javax.swing.JButton ivjOKButton = null;
}
```

When you click the List Customers button on the Order Entry window, the constructor for SLtCustWdw is executed. The constructor receives a reference to the Order Entry window and SQL connection object. Here is the code for the constructor:

```
public SLtCustWdw(OrderEntryWdw orderWdw, Connection dbConnect)
{
    super();
    initialize();
    orderWindow = orderWdw;
    this.dbConnect = dbConnect;
    this.show();
}
```

When the Customer window is opened, the window opening action is connected to the invocation of the instance of a DefaultTableModel object and the column header is set by a returned vector from the returnColumnVector() method. This code is generated by VisualAge for Java:

```

private void connToM2(java.awt.event.WindowEvent arg1) {
    try {
        // user code begin {1}
        // user code end
        getDefaultTableModel1().setColumnIdentifiers(this.returnColumnVector());
        // user code begin {2}
        // user code end
    } catch (java.lang.Throwable ivjExc) {
        // user code begin {3}
        // user code end
        handleException(ivjExc);
    }
}

```

The DefaultTableModel object is used to set the model of the JTable object. This code is also generated by VisualAge for Java because of the connection of these two properties:

```

private void connPtoP1SetTarget() {
    /* Set the target from the source */
    try {
        getScrollPaneTable().setModel(getDefaultTableModel1());
        getScrollPaneTable().createDefaultColumnsFromModel();
        // user code begin {1}
        // user code end
    } catch (java.lang.Throwable ivjExc) {
        // user code begin {3}
        // user code end
        handleException(ivjExc);
    }
}

```

Data retrieval is done and filled in the DefaultTableModel object in the populateCustBox() method. Here is the code for the method:

```

private void populateCustBox()
{
    orderWindow.updateStatus("Retrieving customer list...");

    // The result set that is returned represents records that
    // have 9 fields of data. These fields are all character
    // data and will be stored in an array of strings

    ResultSet rs = null;
    CallableStatement callableStatement = null;

    try
    {
        // invoke the stored procedure on the iSeries
        callableStatement = dbConnect.prepareCall("CALL APILIB.SLTCUSTR(' ','R')");
        rs = callableStatement.executeQuery();
        // each record is fetched from the result set, the fields
        // are retrieved by name and stored in the array
        while (rs.next())
        {
            String[] array = new String[9];
            array[0] = rs.getString("CID");
            array[1] = rs.getString("CLAST");
            array[2] = rs.getString("CFIRST");
            array[3] = rs.getString("CINIT");
            array[4] = rs.getString("CADDR1");
            array[5] = rs.getString("CADDR2");

```

```

        array[6] = rs.getString("CCITY");
        array[7] = rs.getString("CSTATE");
        array[8] = rs.getString("CZIP");
        getDefaultTableModel1().addRow(array);
    }
}
catch (SQLException e)
{
    orderWindow.updateStatus("Error retrieving customer list");
    handleException(e);
    return;
}
orderWindow.updateStatus("Customer list retrieved");
return;
}

```

As previously shown, populateCustbox() receives a result set through an invocation of a remote stored procedure. This is done by running the prepareCall() method, which returns a CallableStatement object.

The executeQuery() method of this object runs the stored procedure:

```
callableStatement = dbConnect.prepareCall("CALL APILIB.SLTCUSTR(' ','R')");
rs = callableStatement.executeQuery();
```

The stored procedure on the iSeries server is called SLTCUSTR. It calls an RPG program that is also named SLTCUSTR. It accepts two parameters. Each parameter is one character long. When the SLTCUSTR program receives two parameters, it bypasses its own display processing and returns a result set. The actual values of the parameters (' ' and 'R' in this case) are arbitrary. They can be set to any character value. The important fact is that two parameters are being passed. This is also covered later when we explain the RPG code.

After the customer table is filled, the user clicks to select a valid customer and then clicks the **OK** button. This action is connected to the invocation of the custSelected() method. An integer value of selected row is passed as a parameter. Here is the code for this method:

```

public void custSelected(int selectedRow) {
    // declare an array of String the same size as the number of columns in
    // the table
    String[] custInf = new String[getScrollPaneTable().getColumnCount()];

    // retrieve the data from the selected row. Each column will be
    // converted to a String object.
    for (int i=0; i<getScrollPaneTable().getColumnCount(); i++)
    {
        custInf[i] = (String) getScrollPaneTable().getValueAt(selectedRow, i);
    }

    // instantiate a Customer object, passing the constructor the array of
    // String data
    Customer aCustomer = new Customer(custInf);

    // invoke the method that will set the text fields
    // in the OrderEntryWdw
    orderWindow.setSelectedCust(aCustomer);

    // close down
    this.dispose();
    return;
}

```

As shown in this code snippet, the `custSelected()` method creates a `Customer` object once a row has been selected from the `JTable` object.

Now, we examine the `Customer` class:

```
public class Customer
{
    private java.math.BigDecimal id;
    private String lastName;
    private String firstName;
    private String init;
    private String address1;
    private String address2;
    private String city;
    private String state;
    private String postCode;
}
```

The `Customer` class is an object-oriented representation of a customer record. The constructor simply sets the data members based on the string elements in the array that is passed in:

```
public Customer ( String[] custInfo)
{
    // parse the array into the appropriate data members
    id = new java.math.BigDecimal(custInfo[0]);
    lastName = custInfo[1];
    firstName = custInfo[2];
    init = custInfo[3];
    address1 = custInfo[4];
    address2 = custInfo[5];
    city = custInfo[6];
    state = custInfo[7];
    postCode = custInfo[8];
}
```

The `Customer` class also provides the standard “getter” methods for retrieving the values of individual data members. These methods are basic and are not discussed here.

As noted previously, the selected `Customer` is passed as a parameter to the `setSelectedCust()` method of the `OrderEntryWdw` object. Here is this method:

```
public void setSelectedCust(Customer selectedCust)
{
    getCustIdTF().setText(selectedCust.getId().toString());
    getCustNameTF().setText(selectedCust.getFullName());
    getStreetTF().setText(selectedCust.getAddress());
    getCityTF().setText(selectedCust.getCity());
    getStateTF().setText(selectedCust.getState());
    getPCodeTF().setText(selectedCust.getPostCode());
    getListItemBTN().setEnabled(true);
    this.requestFocus();
    updateStatus("Customer information set");
    return;
}
```

The method simply retrieves values from the `Customer` object and sets the appropriate text fields in the Order Entry window. Figure 8-8 shows the current state of the main window.

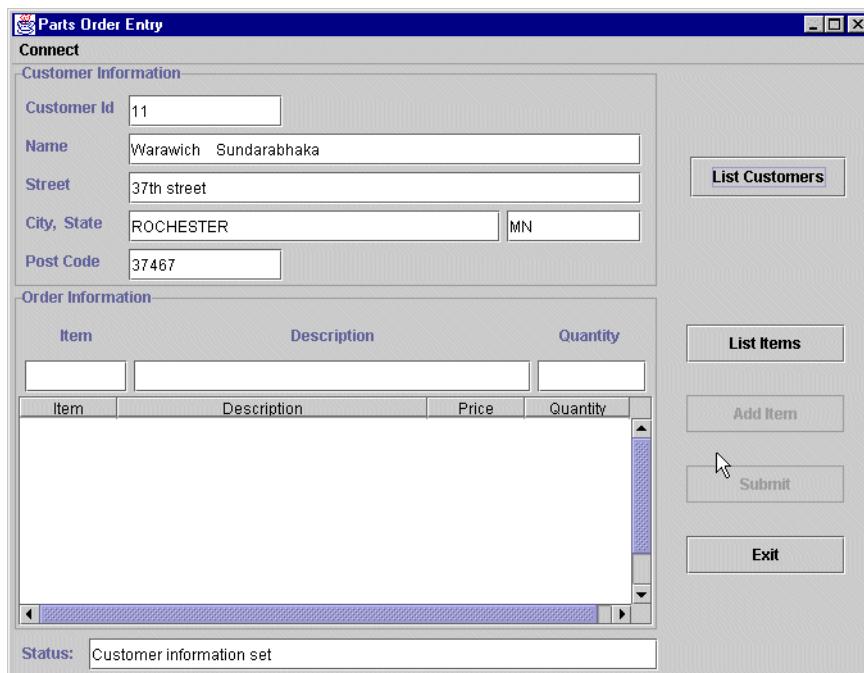


Figure 8-8 Order Entry window with customer data

Once the information for the selected customer is set, the List Items button is enabled and we are ready to retrieve the list of valid items from the iSeries server.

#### 8.4.4 Retrieving the item list

This section examines the code that is needed to retrieve a list of valid items from the iSeries server. The process is similar to retrieving the customer list. Figure 8-9 shows the window that is displayed once the list of items is retrieved.

Part #	Description	Price	Quantity
000001	WEBSPHERE REDBOOK	\$30.00	317
000002	Radio_Controlled_Plane	\$96.86	7
000003	Change_Machine	\$52.55	37
000004	Baseball_Tickets	\$91.14	899
000005	Twelve_Num_Two_Pencils	\$50.58	1715
000006	Over_Under_Shotgun	\$79.66	1306
000007	Feel_Good_Vitamins	\$29.81	200
000008	Cross_Country_Ski_Set	\$93.00	54
000009	Rubber_Baby_Buggy_Wheel	\$98.71	200
000010	ITSO REDBOOK SG24-2152	\$50.00	297
000011	ITSO REDBOOK SG24-2163	\$20.87	982
000012	Plastic_Garbage_Pail	\$22.40	84
000013	Doll_House_Furniture	\$24.22	541

Figure 8-9 Select Items

SltItemWdw is the class that defines this window. It contains a JTable object and one button. The JDBC interface is used to access an SQL stored procedure on the iSeries server. This stored procedure returns a result set. Records are fetched from the result set and the retrieved data populates the table. This code snippet shows the class definition for SltItemWdw:

```

import java.sql.*;
import orderObjects.*;

public class SltItemWdw extends javax.swing.JFrame implements
    java.awt.event.ActionListener, java.awt.event.WindowListener{
    private Connection dbConnect = null;
    private OrderEntryWdw orderWindow = null;
    private javax.swing.JPanel ivjPanel1 = null;
    private javax.swing.JPanel ivjJPanel1 = null;
    private javax.swing.JScrollPane ivjScrollPane1 = null;
    private javax.swing.JButton ivjOKButton = null;
    private javax.swing.JTable ivjScrollPaneTable = null;
    private javax.swing.table.DefaultTableModel ivjDefaultTableModel1 = null;
}

```

When you click the List Items button on the Order Entry window, the constructor for SltItemWdw is executed. The constructor receives a reference to the Order Entry window and a reference to the SQL connection object. Here is the code for the constructor:

```

public SltItemWdw(OrderEntryWdw orderWdw, Connection dbConnect)
{
    super();
    initialize();
    orderWindow = orderWdw;
    this.dbConnect = dbConnect;
    this.show();
}

```

When the Item window is opened. The window opening action is connected to the invocation of the instance of a DefaultTableModel object and the column header is set by a returned vector from the returnColumnVector() method. This code is generated by VisualAge for Java.

```

private void connEtoM3(java.awt.event.WindowEvent arg1) {
    try {
        // user code begin {1}
        // user code end
        getDefaultTableModel1().setColumnIdentifiers(this.returnColumnVector());
        // user code begin {2}
        // user code end
    } catch (java.lang.Throwable ivjExc) {
        // user code begin {3}
        // user code end
        handleException(ivjExc);
    }
}

```

The DefaultTableModel object is used to set the model of the JTable object. This code is also generated by VisualAge for Java because of the connection of these two properties.

```

private void connPtoP1SetTarget() {
    /* Set the target from the source */
    try {
        getScrollPaneTable().setModel(getDefaultTableModel1());
        getScrollPaneTable().createDefaultColumnsFromModel();
        // user code begin {1}
        // user code end
    } catch (java.lang.Throwable ivjExc) {
        // user code begin {3}
        // user code end
        handleException(ivjExc);
    }
}

```

Data retrieval is done and filled in the DefaultTableModel object in the populateItemBox() method. Here is the code for the method:

```
private void populateItemBox()
{
    orderWindow.updateStatus("Retrieving item list...");

    // The result set that is returned represents records that
    // have 4 fields of data. These fields will be stored in
    // an array of strings

    ResultSet rs;
    CallableStatement callableStatement;

    try
    {
        // invoke the stored procedure on the iSeries
        callableStatement = dbConnect.prepareCall("CALL APILIB.SLTPARTR(' ','R')");
        rs = callableStatement.executeQuery();
        while (rs.next()) {
            String[] array = new String[4];
            array[0] = rs.getString("IID");
            array[1] = rs.getString("INAME");
            array[2] = "$" + rs.getBigDecimal("IPRICE").toString();
            array[3] = Integer.toString(rs.getInt("STQTY"));
            getDefaultTableModel1().addRow(array);
        }
    }
    catch (SQLException e)
    {
        orderWindow.updateStatus("Error retrieving item list");
        handleException(e);
        return;
    }
    orderWindow.updateStatus("Item list retrieved");
    return;
}
```

The process of populating the item table is almost identical to the process of populating the customer table. However, two of the fields that are retrieved are not characters. The IPRICE field (column) is stored as packed decimal data on the iSeries server. It is mapped to a Java BigDecimal type with two decimal positions in the Java code.

This result is then converted to a String:

```
array[2] = "$"+rs.getBigDecimal("IPRICE",2).toString();
```

The STQTY field is mapped to a Java type of Integer since there are no decimal positions involved. The string value of this is put into the array. To do this, execute the static `toString()` method of the Integer class:

```
array[3] = Integer.toString(rs.getInt("STQTY"));
```

After the item table is filled, the user clicks to select a valid item and clicks the OK button. This action is connected to the invocation of the `itemSelected()` method. An integer value of selected row is passed as a parameter. Here is the code for this method:

```
public void itemSelected(int selectedRow)
{
    // declare an array of String the same size as the number of columns in
    // the table
    String[] itemInf = new String[getScrollPaneTable().getColumnCount()];
```

```

// retrieve the data from the selected row. Each column will be
// converted to a String object.
for (int i=0; i<getScrollPaneTable().getColumnCount(); i++)
{
    itemInf[i] = (String) getScrollPaneTable().getValueAt(selectedRow, i);
}

// instantiate an Item object, passing the constructor the array of
// String data
Item aItem = new Item(itemInf);

// invoke the method that will set the text fields
// in the OrderEntryWdw
orderWindow.setSelectedItem(aItem);

// close down the list window
this.dispose();
return;
}

```

The `itemSelected()` method retrieves the selected row, converts it to String, and adds it as an element in a String array. It then creates an ITEM object and passes this to the `setSelectedItem()` method.

```

public class Item
{
    private String id;
    private String name;
    private java.math.BigDecimal price;
    private int quantity;
}

```

The ITEM class is an object-oriented representation of a record in the ITEM file. The constructor takes a String array as a parameter and then sets the data members accordingly:

```

public Item ( String[] itemInf)
{
    id = itemInf[0];
    name = itemInf[1];
    // remember to trim the '$' symbol before
    // instantiating a BigDecimal

    price = new java.math.BigDecimal(itemInf[2].substring(1));
    quantity = new Integer(itemInf[3]).intValue();
}

```

The `setSelectedItem()` method in the OrderEntryWdw class puts the ITEM ID and the ITEM name in the window. It positions the cursor to the quantity field, where a number must be entered before the entry is added to the order list:

```

public void setSelectedItem(Item selectedItem)
{
    getItemTF().setText(selectedItem.getId());
    getDscTF().setText(selectedItem.getName());
    getQtyTF().requestFocus();
    this.requestFocus();
    updateStatus("Item information set: please enter quantity");
    return;
}

```

#### 8.4.5 Verifying and adding the item to the order

Once a quantity is entered, the Add Item button is enabled. The action of this button is connected to an invocation of the addOrderItem() method. The value in the item text field is passed in as a parameter.

This method uses the RECORDACCESS functionality that the IBM Toolbox for Java provides to retrieve a record from the ITEM file. If the record is found, it adds the information to the order table in the main window and enables the Submit button. If the record is not found or if there is an error reading the file, the status field is updated and the Submit button is not enabled. This acts as a verification for the item that is being ordered (in case an item ID and quantity are incorrectly entered without using the prompt function offered by the List Items button). Here is the code for the method:

```
public void addOrderItem(String key)
{
    // This method is invoked when the add item button
    // is pressed. It gets the text from the item text
    // field (getItemTF().getText()) and uses it as a
    // key to read the ITEM file.

    Record data = null;
    Object[] theKey = new Object[1];
    theKey[0] = key;
    updateStatus("Verifying order item...");

    if (itemFile == null)
    {
        openItemFile();
    }
    try
    {
        data = itemFile.read(theKey);
    }
    catch (Exception e)
    {
        updateStatus("Error reading ITEM file");
        handleException(e);
        return;
    }
    try
    {
        if (data != null)
        {
            // retrieve data from the record and build an
            // entry in the order box
            String[] orderRow = new String[4];
            orderRow[0] = data.getField("IID").toString();
            orderRow[1] = data.getField("INAME").toString();
            orderRow[2] = data.getField("IPRICE").toString();
            orderRow[3] = getQtyTF().getText();
            getDefaultTableModel1().addRow(orderRow);
            getSubmitBTN().setEnabled(true);
            updateStatus("Item added: please add more or submit");
        }
        else
        {
            updateStatus("Invalid item...");
        }
    }
}
```

```

        catch (Exception e)
    {
        updateStatus("Error retrieving field data from Item file");
        handleException(e);
        return;
    }
}

```

Once an item is added to the order table, the Submit button is enabled. More items may be added to the list, or the order may be submitted. Figure 8-10 shows the state of the window at this point.

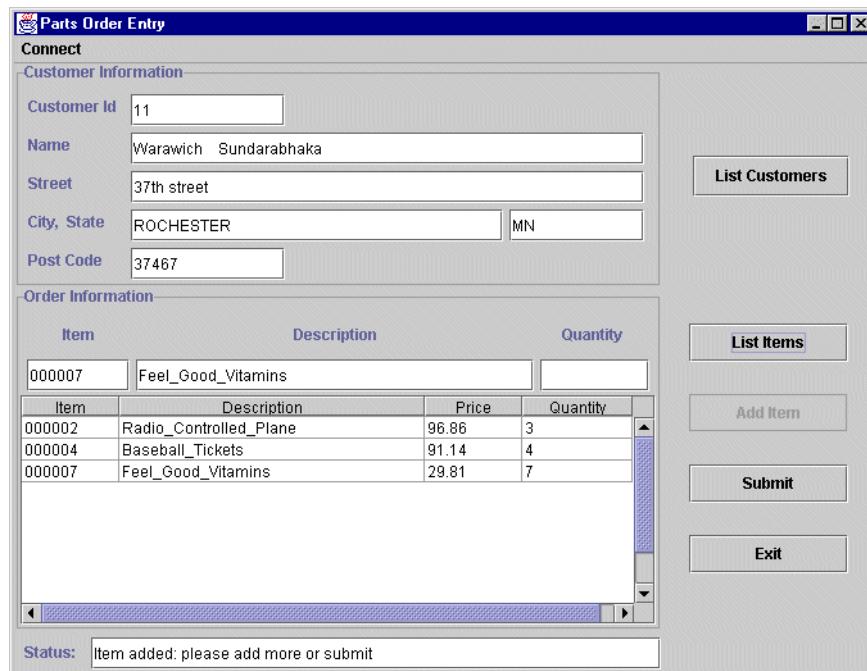


Figure 8-10 Parts Order ready to submit

#### 8.4.6 Submitting the order

When you click the Submit button, the `retrieveOrderInfo()` method is called. This method retrieves the order information from the window. It constructs an `Order` object, passing the customer ID to the constructor. It then adds `OrderDetail` objects to the `Order` object by retrieving each row from the order table.

```

private void retrieveOrderInfo()
{
    int numEntries = getDefaultTableModel1().getRowCount();
    Order theOrder = new Order(getCustIdTF().getText());

    for(int i=0;i<numEntries;i++)
    {
        OrderDetail detail = new OrderDetail(
            (String)getDefaultTableModel1().getValueAt(i,0),
            (String)getDefaultTableModel1().getValueAt(i,1),
            new BigDecimal((String)getDefaultTableModel1().getValueAt(i,2)),
            new BigDecimal((String)getDefaultTableModel1().getValueAt(i,3)));
        theOrder.addEntry(detail);
    }
    submitOrder(theOrder);
}

```

```
        return;
    }
```

The Order class is now examined:

```
public class Order
{
    private String customerId;
    private java.util.Vector orderDetail = new java.util.Vector();
}
```

As shown, an Order object contains a vector of OrderDetail objects. The customerId field is declared as a String. The iSeries server program that is eventually called expects a buffer of character data. Certain fields need to have specific lengths, so offsets are predictable. The customerId field must always be a length of 4. If the customerId retrieved from the window is 10, then two leading zeros must be inserted to yield 0010. The leadingZero() static method is used to perform this conversion. This is shown in the constructor:

```
public Order ( String cid)
{
    // Set the customer id making sure leading zeros are
    // included.
    customerId = Order.leadingZero(cid, 4);
}
```

Here is the code of the leadingZero() method. The first parameter is the original string, and the second parameter is the length of the returned string.

```
public static String leadingZero(String aString, int len)
{
    StringBuffer buf = new StringBuffer(len);
    for(int i=0; i<len-aString.length(); i++)
    {
        buf.append('0');
    }
    return buf.append(aString).toString();
}
```

Since the iSeries server program that processes orders (ORDENTR) expects all character data, a `toString()` method is provided in the Order class. This method converts the Order object into one contiguous string. The string may be viewed as a buffer with the following breakdown:

- ▶ Starting at offset 0 for a length of 4 bytes: The customer ID
- ▶ Starting at offset 4 for a length of 5 bytes: The number of detail entries
- ▶ Starting at offset 9 with varying length: Multiple 40-byte segments

Each 40-byte segment represents a single detail record that consists of an item ID, name, price, and quantity. See the `OrderDetail.toString()` method for a granular breakdown of a detail record.

This is the `toString()` method for the Order class:

```
public String toString()
{
    // declare a StringBuffer
    StringBuffer orderBuffer =
        new StringBuffer(9+(40*getOrderDetail().size()));
    // append the customerId to the buffer
    orderBuffer.append(customerId);

    // convert the number of entries to a string of 5 bytes
    // ...
}
```

```

// and be sure to include leading zeros
String numEntryString = leadingZero(Integer.toString(getOrderDetail().size()), 5);

// now append the number of entries string to the order buffer
orderBuffer.append(numEntryString);

// Get Order detail elements from the vector
java.util.Enumeration e = orderDetail.elements();
while(e.hasMoreElements())
{
    OrderDetail anOrderDetail = (OrderDetail) e.nextElement();
    // append a string representation of the detail entry
    // this is done by invoking the toString() method that
    // is provided by the OrderDetail class
    orderBuffer.append(anOrderDetail.toString());
}

// return the complete buffer as a string
return orderBuffer.toString();
}

```

The OrderDetail class is shown here:

```

import java.math.*;

public class OrderDetail
{
    String itemId;
    String itemDsc;
    BigDecimal itemPrice = null;
    BigDecimal itemQty = null;
}

```

As in the Order class, take care so that certain data members have a specific length. In some cases, the constructor must add leading zeros to the itemId by calling the leadingZero() static method in the Order class. It may also have to add trailing blanks to itemDsc. Here is the constructor:

```

public OrderDetail ( String itemId, String itemDsc, BigDecimal itemPrice,
                     BigDecimal itemQty)
{
    // set the itemId making sure leading zeros are there
    this.itemId = Order.leadingZero(itemId, 6);

    // set the description making sure trailing blanks are there
    this.itemDsc = itemDsc;
    for(int i=itemDsc.length();i<24;i++)
    {
        this.itemDsc += " ";
    }

    this.itemPrice = itemPrice;
    this.itemQty = itemQty;
}

```

The OrderDetail class also provides a `toString()` method. Once again, this facilitates the call to the iSeries server program that accepts parameters as character data. The `toString()` method converts the orderDetail object to a buffer of 40 characters with certain offsets that are designated as starting points of certain data members. The breakdown of the orderDetail buffer is:

- ▶ Starting at offset 0 for 6 bytes: The item ID
- ▶ Starting at offset 6 for 24 bytes: The item name (description)
- ▶ Starting at offset 30 for 5 bytes: The price
- ▶ Starting at offset 35 for 5 bytes: The quantity ordered

```

public String toString()
{
    StringBuffer entryBuffer = new StringBuffer(40);
    entryBuffer.append(itemId);
    entryBuffer.append(itemDsc);

    // convert price field to String, remove
    // decimal point, and make sure leading
    // zeros are there
    String priceString = itemPrice.toString();
    int dec = priceString.indexOf('.');
    priceString = priceString.substring(0,dec)+priceString.substring(dec+1);
    priceString = Order.leadingZero(priceString, 5);

    // now append it to the entry buffer
    entryBuffer.append(priceString);

    // convert the quantity field to String and make sure
    // it is 5 bytes. Then append it to the entry buffer
    entryBuffer.append(Order.leadingZero(itemQty.toString(), 5));

    // now return the whole entry as a String
    return entryBuffer.toString();
}

```

Previously we mentioned that the action of the Submit button was connected to an invocation of the retrieveOrderInfo() method. That method creates an Order and adds OrderDetail objects to it. Next, the retrieveOrderInfo() method executes the submitOrder() method and passes the Order object as a parameter. The SubmitOrder() method calls the ORDENTR program on the iSeries server using the Distributed Program Call class that is provided by the IBM Toolbox for Java. When running the ORDENTR RPG program that processes orders, this program requires two parameters.

The first parameter is a string that is a concatenation of the customer ID and the number of entries in the order. This data is fixed in length. The first 4 bytes are designated for the customer ID, while the last 5 bytes are for the number of entries. The ORDENTR program moves the customer ID data into a character field that has a length of 4 bytes. The last 5 bytes are moved to a zoned numeric field. These 9 bytes of data are the first 9 bytes in the string that is returned by the Order.toString() method.

The second parameter is a block of character data that represents all of the detail entries in the order. This data is sent as one contiguous block of character data to the ORDENTR program that parses the data. This block of data is also part of the string that is returned by the Order.toString() method. It begins at offset 9 of the string.

This is the submitOrder() method:

```

public void submitOrder(Order theOrder)
{
    updateStatus("Processing...if you hang here check QZRCRSVS");

    String orderString = theOrder.toString();
    String header = orderString.substring(0,9);
    String detail = orderString.substring(9);

```

```

try
{
   getAs400().connectService(AS400.COMMAND);
    ProgramCall ordEntrPgm = new ProgramCall(getAs400());
    QSYSObjectPathName pgmName = new QSYSObjectPathName("APILIB","ORDENTR","PGM");

    ProgramParameter[] parmList = new ProgramParameter[2];

    // set the first parameter which is the order header
    AS400Text text1 = new AS400Text(9, getAs400());
    byte[] headerInfo = text1.toBytes(header);
    parmList[0] = new ProgramParameter(headerInfo);

    // set the second parameter which is the order detail
    AS400Text text2 = new AS400Text(detail.length(), getAs400());
    byte[] detailInfo = text2.toBytes(detail);
    parmList[1] = new ProgramParameter(detailInfo);

    ordEntrPgm.setProgram(pgmName.getPath(),parmList);

    // if you hang here - the iSeries server job (QZRCSRVS)
    // may be waiting on an inquiry message and control
    // will never return. Check the QZRCSRVS joblog
    if (ordEntrPgm.run() != true)
    {
        // If you get here, the program failed to run.
        // Get the list of messages to determine why
        // the program didn't run.
        AS400Message[] messageList = ordEntrPgm.getMessageList();
        updateStatus(messageList[0].getText());
    }
    else
    {
        updateStatus("Order successfully submitted");
    }
}
catch(Exception e)
{
    updateStatus("Error submitting order");
    handleException(e);
}
return;
}

```

Now, we analyze the method.

In order to supply parameters for the ORDENTR program on the iSeries server, we extract the parameters from the String object that is returned by the Order.toString() method as header and detail string respectively.

We start the AS400.COMMAND service for the as400 object. We refer to as400 object by calling its accessor, the getAs400() method.

```

public com.ibm.as400.access.AS400 getAs400()
{
    if(as400==null)
    {
        as400 = new AS400(this.systemName, this.userid, this.password);
    }
}

```

```
        return as400;
    }
```

**Note:** The IBM Toolbox for Java implicitly starts this if it needs to be started. It is shown here for clarification:

```
getAs400().connectService(AS400.COMMAND);
```

We declare a ProgramCall object called ordEntrPgm. Then, we set the name of the iSeries server program associated with this object by declaring and initializing a QSYSObjectPathName object:

```
ProgramCall ordEntrPgm = new ProgramCall(getAs400());
QSYSObjectPathName pgmName = new QSYSObjectPathName("APILIB","ORDENTR","PGM");
```

Once this is done, you must set the parameters for this program. To set the parameters, perform these steps:

1. Declare an array of ProgramParameter objects. We declare an array of two, since the program requires two parameters, by entering:

```
ProgramParameter[] parmList = new ProgramParameter[2];
```

2. Construct the individual ProgramParameter elements to fill the array. The ProgramParameter constructor must be passed an array of bytes. Before instantiating a ProgramParameter object, we must first generate the appropriate array of bytes. To create the array of bytes:

- a. Declare an appropriate iSeries server data type object. In this case, we are passing string data, so we declare an AS400Text object.
- b. Execute the toBytes() method on this AS400Text object and pass the string that is being converted to bytes. Here are the two steps that you need:

```
AS400Text text1 = new AS400Text(header.length(), getAs400());
byte[] headerBytes = text1.toBytes(header);
```

3. Execute the constructor for the ProgramParameter class by entering:

```
parmList[0] = new ProgramParameter(headerBytes);
```

Now, the same scenario is followed for each additional parameter that you need. Once the parameters have been instantiated, the ProgramCall object must be initialized with the actual name of the iSeries program that is being called. The parameter list must also be specified by executing the setProgram() method:

```
ordEntrPgm.setProgram(pgmName.getPath(),parmList);
```

We are now ready to run the program. The run() method executes the program. In this example, it is done inside an "if" construct, so any errors may be processed:

```
if (ordEntrPgm.run() != true)
{
    // If you get here, the program failed to run.
    // Get the list of messages to determine why
    // the program didn't run.
    AS400Message[] messageList = ordEntrPgm.getMessageList();
    updateStatus(messageList[0].getText());
}
```

One important fact should be noted here. If the program on the iSeries server issues a message that waits for a response (an inquiry message), then *control is never returned*. The submitOrder() method hangs on the ordEntrPgm.run() method. If this occurs, you must check the server job that is handling the request on the host iSeries server. This job has a name of QZRCSRVS, and its job log should be viewed. If the program runs successfully, the submitOrder() method updates the status text field accordingly.

This concludes the application flow from the Java client perspective. Now, we examine the code changes made in the RPG code to accommodate the Java client.

## 8.5 Changes to the host Order Entry application

This section contains details about the transition of the RPG code on the host. The changes are made, so the application can run in one of two modes:

- ▶ As a native application with a 5250 screen interaction
- ▶ In conjunction with the newly created Java client

For the most part, the changes are examined in the same sequence as the client application flow.

### 8.5.1 Providing a customer list

We saw earlier that one of the first things the client does after connecting is to request a customer list. This was done by executing an SQL stored procedure using JDBC. The stored procedure is actually an RPG program called SLTCUSR. To accommodate the Java client, two subroutines are added to the program and the logic flow is changed when a second parameter is detected. In the original version of SLTCUSR, the logic flow can be summarized as:

- ▶ Run the **OpenCust** subroutine (declares and opens the cursor for CSTMR file).
- ▶ Run the **BldSfl** subroutine (populates and displays the subfile).
- ▶ Run the **Process** subroutine (detects the chosen customer and displays).

The new logic flow can be summarized as:

- ▶ Run the **CloseCust** subroutine (resets the cursor for multiple client requests).
- ▶ Run the **OpenCust** subroutine (same behavior as the original version).
- ▶ If more than one parm, run the **ResultSet** subroutine (makes result set available to the caller of the stored procedure).
- ▶ If only one parm, continue as the original version did. Run the **BldSfl** subroutine, and then run **Process**.

The added subroutines are minimal code changes. Here is the CloseCust subroutine:

```
CSR  CloseCust      BEGSR
  *
  -----
C/Exec Sql Close CUSTOMER
C/End-Exec
  *
CSR          ENDSR
```

As previously mentioned, this ensures that the cursor is at the beginning of the file when multiple requests are received from the client. Now, we examine the ResultSet subroutine:

```

CSR  ResultSet      BEGSR
* -----
C/Exec Sql
C+ Set Result Sets Cursor CUSTOMER
C/End-Exec
*
CSR          ENDSR

```

This allows the client to retrieve the result set when the program is called as a stored procedure.

The program determines the number of parameters by accessing a pre-defined field in the Program Status Data Structure:

```

D PgmstsDS      SDS
D NbrParms      *PARMS

```

After executing the OpenCust subroutine, the program determines if more than one parameter was passed. If this is the case, the ResultSet subroutine runs, and control is returned. The subfile processing is bypassed:

```

C           IF      NbrParms > 1
C           EXSR    ResultSet
C           RETURN
C           ENDIF

```

The file specifications for the display file are changed, so user controlled open is specified (the keyword USROPN is added). There is no reason to open the file if the program is executed from a Java client. The partial line is shown here:

```
...WORKSTN SFILE(CUSTSFL:Sf1Rrn) USROPN
```

It is interesting to note that the functionality handled by the BldSfl subroutine is analogous to the processing done by the S1tCustWdw.populateCustBox() method. The functionality handled by the Process subroutine is handled by the S1tCustWdw.custSelected() and OrderEntryWdw2.setSelectedCust() methods.

## 8.5.2 Providing an item list

As you saw earlier, the List Items button executes the SLTPARTR stored procedure. The changes made in SLTPARTR to accommodate the Java client are similar to the changes made in SLTCUSTR. The original logic flow in the SLTPARTR program is virtually identical to the flow in SLTCUSTR:

- ▶ Run the **OpenPart** subroutine (declares and opens the cursor for ITEM and STOCK file).
- ▶ Run the **BldSfl** subroutine (populates and displays the subfile).
- ▶ Run the **Process** subroutine (detects the chosen part or item and displays it).

The new logic flow can be summarized as:

- ▶ Run the **ClosePart** subroutine (resets the cursor for multiple client requests).
- ▶ Run the **OpenPart** subroutine (same behavior as the original version).
- ▶ If more than one parm, run the **ResultSet** subroutine (makes the result set available to the caller of the stored procedure).
- ▶ If only one parm, continue as the original version did. Run the **BldSfl** subroutine, and then run **Process**.

Since the code changes are virtually identical to the ones made in SLTCUSTR, they are not discussed here but can be downloaded and viewed.

### 8.5.3 Verifying an item

From the client, an item is verified using the direct RECORDACCESS capability that the IBM Toolbox for Java offers. Since no host RPG program is used, no changes are involved in this process. The only task left to do is handle an order submitted from the client.

### 8.5.4 Processing the submitted order

As previously discussed, the iSeries server program that handles a request to submit an order is ORDENTR. When the Order Entry application is run from an iSeries server 5250 session (no Java client), ORDENTR is the entry point of the application. It displays the initial windows that correspond to the Order Entry window in the Java client version. The ORDENTR program must be changed, so it recognizes the fact that it is executed from Java.

First, the number of parameters are ascertained through the program status data structure:

```
D PgmStsDS      SDS
D NbrParms      *PARMS
```

If the number of parameters is greater than zero, it is assumed that the program has been executed as a distributed program.

Since the Java client passes in two parameters, two data structures are declared that map to the parameters. As discussed earlier, the client passes two strings. The first string is 9 characters representing the customer ID (4 characters), and the number of detail entries (5 characters).

A data structure named CustDS is declared for this first parameter:

```
D CustDS        DS
D   CustNbr          LIKE(CID)
D   OrdLInCnt       5  0
```

The second parameter is a string that represents a contiguous grouping of detail entries. Each entry has a length of 40, and there are a maximum of 50 entries. A data structure named OrderMODS is declared for this parameter.

```
D OrderMODS     DS          OCCURS(50)
D   PartNbrX      LIKE(IID)
D   PartDscX      LIKE(INAME)
D   PartPriceX    5  2
D   PartQtyX      5  0
```

An entry parameter list is added to the initialization subroutine. This ensures that the data structures are loaded with the parameter values passed in:

```
C   *ENTRY      PLIST
C           PARM      CustDS
C           PARM      OrderMODS
```

As in the other RPG programs, the USROPN keyword is added to the file specification, since the file is not opened when started as a distributed program. Here is the portion of the file specification with the USROPN keyword added:

```
...WORKSTN SFILE(ORDSFL:Sf1Rrn) USROPN
```

The mainline logic of the program is changed to check the number of parameters. If there are parameters, a new subroutine, called CmtOrder2, executes and all display file processing is bypassed:

```

C           IF      NbrParms > *ZERO
C           EXSR    CmtOrder2
C           EXSR    EndPgm
C           ENDIF

```

The CmtOrder2 subroutine is similar to the original CmtOrder subroutine. However, it retrieves the order information from the CustDS and OrderMODS data structures, rather than from the display file and subfile records:

```

CSR  CmtOrder2      BEGSR
* -----
* Get the next order number
C           EXSR    GetOrdNbr
*
* Get the order date and time
C           TIME          DateTime
C           Z-ADD     *ZERO      OrdTot
*
* Get the customer information
C           MOVE      CustNbr    CustomerId
C   CustKey   CHAIN    CSRCD
*
* For each order line in the passed structure ...
C   1        DO      OrdLinCnt  OrdCnt
C   OrdCnt   OCCUR   OrderMODS
* Set up the fields so the existing DB routines work
C           MOVE      PartNbrX   PARTNBR_0
C           MOVE      PartDscX   PARTDSC_0
C           MOVE      PartQtyX   PARTQTY_0
C           MOVE      PartPriceX ITEMPRICE
* Add an order detail record ...
C           EXSR    AddOrdLin
* Update stock record ...
C   StockKey  CHAIN    STRCD
C           EXSR    UpdStock
* Accumulate order total ...
C           EVAL    OrdTot = OrdTot + OLAMNT
C           ENDDO
*
* Add an order header record ...
C           EXSR    AddOrdHdr
*
* Update customer record ...
C           EXSR    UpdCust
*
* Commit the database changes ...
C           IF      CmtActive = $True
C           COMMIT
C           ENDIF
*
* Request batch print server to print order
C           EXSR    WrtDtaQ
*
CSR           ENDSR

```

The subroutine is built, so all other existing subroutines can be used as in the prior version. Once again, the only significant change is that the information for the order is retrieved from the parameters that are passed in.

## 8.6 Summary

The common thread pervasive across all the changes in the host RPG code deals with display file processing. When ORDENTR executes as a distributed program, all display file processing is bypassed. The information normally received from the display files and subfiles is now made available through parameters.

Different approaches can be taken. The scenario shown here is not the only valid one. For example, the detail order entries can be passed to the iSeries server as data queue entries. However, this approach entails more changes in the host application. The amount of change needed at the host end is largely affected by design decisions made at the Java client end.

Of course, this only covers certain portions of migrating the application to Java. The server code can also be converted. This topic is covered in Chapter 9, “Moving the server application to Java” on page 349.



# Moving the server application to Java

This chapter discusses changing the server application to Java. We use remote method invocation (RMI) to communicate between a client program and the server program.

## 9.1 What RMI is

Distributed systems require computations running in different address spaces, potentially on different hosts, to communicate. For a basic communication mechanism, the Java language supports sockets, which are flexible and sufficient for general communication. However, sockets require the client and server to engage in application-level protocols to encode and decode messages for exchange. The design of such protocols is cumbersome and can be error prone.

An alternative to sockets is Remote Procedure Call (RPC), which abstracts the communication interface to the level of a procedure call. Instead of working directly with sockets, the programmer has the illusion of calling a local procedure. In fact, the arguments of the call are packaged and shipped to the remote target of the call. RPC systems encode arguments and return values using an external data representation, such as XDR. However, RPC does not translate well into distributed object systems, where communication between program-level objects residing in different address spaces is needed. To match the semantics of object invocation, distributed object systems require remote method invocation. In such systems, a local surrogate (stub) object manages the invocation on a remote object.

The Java remote method invocation system described in this specification has been specifically designed to operate in the Java environment. While other RMI systems can be adapted to handle Java objects, these systems fall short of seamless integration with the Java system due to their interoperability requirement with other languages. For example, CORBA presumes a heterogeneous, multi-language environment, and therefore, must have a language-neutral object model. In contrast, the Java language RMI system assumes the homogeneous environment of the Java virtual machine. Therefore, the system can take advantage of the Java object model whenever possible.

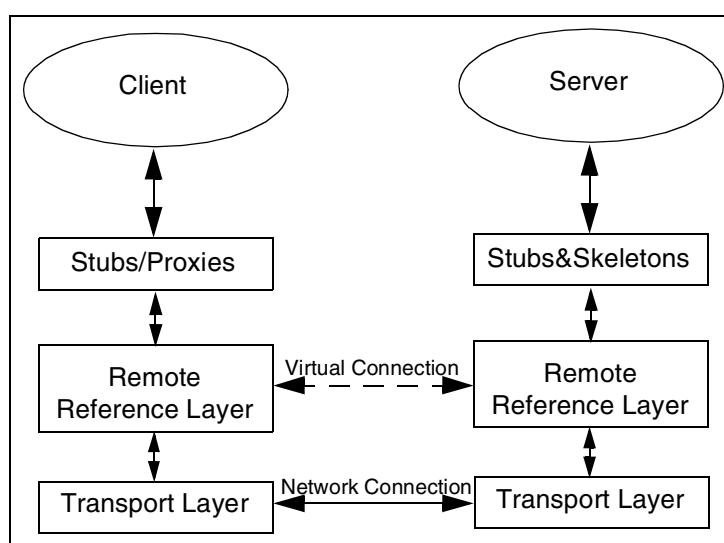


Figure 9-1 RMI architecture

As shown in Figure 9-1, a remote method invocation from a client to a remote server object travels down through the layers of the RMI system to the client-side transport. Then, it travels up through the server-side transport to the server.

A client invoking a method on a remote server object actually makes use of a stub or proxy for the remote object as a conduit to the remote object. A client-held reference to a remote object is a reference to a local stub. This stub is an

implementation of the remote interfaces of the remote object and forwards invocation requests to that server object through the remote reference layer. Stubs are generated using the rmic compiler.

The remote reference layer is responsible for carrying out the semantics of the invocation. For example, the remote reference layer is responsible for determining whether the server is a single object or a replicated object requiring communications with multiple locations. Each remote object implementation chooses its own remote reference semantics, whether the server is a single object or is a replicated object requiring communications with its replicas.

Also handled by the remote reference layer are the reference semantics for the server. The remote reference layer, for example, abstracts the different ways of referring to objects that are implemented in servers that are always running on some machine, and servers that are run only when some method invocation is made on them (activation). These differences are not seen at the layers above the remote reference layer.

The transport layer is responsible for connection setup and connection management. Plus, it keeps track of and dispatching to remote objects (the targets of remote calls) residing in the transport address space.

To dispatch to a remote object, the transport forwards the remote call up to the remote reference layer. The remote reference layer handles any server-side behavior that needs to occur before handing off the request to the server-side skeleton. The skeleton for a remote object makes a call up to the remote object implementation that carries out the actual method call.

The return value of a call is sent back through the skeleton, remote reference layer, and transport on the server side. Then, it travels up through the transport, remote reference layer, and stub on the client side.

## 9.2 Building an RMI application

Building an RMI application is a five-step process that follows this sequence:

1. Define the interfaces to your remote server objects.

A Java Interface is like an abstract class. It allows us to define methods without actually implementing them. We can implement the interface in a class.

2. Implement the remote server objects.

The remote class can implement any number of remote interfaces. The class can extend another remote implementation class. The class can define methods that do not appear in the remote interface. Those methods can only be used locally and are not available remotely.

3. Run the `rmic` command on remote implementation classes.

The `rmic` command creates stubs (proxies) and skeletons. It is available as part of JDK1.1. It is also available a part of many IDEs including VisualAge for Java. In the Java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete. RMI uses reflection to make the connection to the remote service object. You only have to worry about skeleton classes and objects in JDK 1.1 and JDK 1.1 compatible system implementations.

4. Implement the client.

The client invokes the remote interfaces defined by the server.

5. Make the server code network accessible.

The server code is made network accessible by running the RMI registry and executing and registering the server object.

## 9.3 Building a simple iSeries application using RMI

This section shows you how to build a simple iSeries client/server application using RMI. This example helps you understand the basic requirements of RMI. Later, in this chapter, we build a more complex example. This first example allows a client program to invoke a remote method, which increments a parameter passed in and returns the result. We build this application following the previously discussed five-step process.

### 9.3.1 Defining interfaces

A Java interface, shown in Figure 9-2, defines a set of methods, but does not implement them. The class that implements the interface agrees to implement all methods defined in the interface. An interface exposes the programming interface of an object without revealing its class.

To use RMI, the interface must:

- ▶ Be a subclass of the `Remote` class
- ▶ Describe each public method
- ▶ Throw a `RemoteException` for each public method

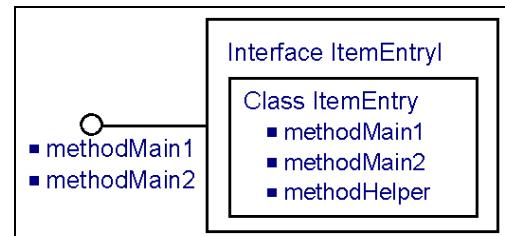


Figure 9-2 Java interfaces

Figure 9-3 shows an interface definition.

```
package TestRMI;  
  
import java.rmi.*; //for Remote, RemoteException  
  
public interface AddOneServerInterface extends Remote {  
  
    public int addOne(int iNum) throws RemoteException;  
}
```

Figure 9-3 Defining the interface

### 9.3.2 Implementing the remote server object

The remote server objects follow these rules:

- ▶ The Class must extend `UnicastRemoteObject` and implement the interface.  
The `UnicastRemoteObject` class defines the remote object as a unicast object, which means that only a single instance of the object can exist on a single server. This is distinguished from a `MultiCastRemoteObject`, which can replicate across multiple servers. The class must implement an interface that describes the public methods.
- ▶ The constructor must throw an Exception.
- ▶ Worker methods must throw a `RemoteException`.
- ▶ The remote object must make its services available by:
  - Binding an instance of the server object to the host
  - The TCP/IP port number used must match the number used on the `rmiregistry` command. We use port 6666 in this example, but you can use any available port.

The host RMI code (Figure 9-4) extends `UnicastRemoteObject` and implements the public interface. The public methods that it implements (`addOne`) throw a `RemoteException`.

```
package TestRMI;

import java.rmi.*;
import java.rmi.server.*;

public class AddOneServer extends UnicastRemoteObject
    implements AddOneServerInterface{

    public AddOneServer() throws Exception {}

    public int addOne(int iNum) throws RemoteException {
        try {
            System.out.println("addOne - someone calling us...");
            System.out.println("iNum = " + iNum);
            return (iNum + 1); // Complex business logic here!
        } catch (Exception e) {e.printStackTrace(); return 0;}
    }
}
```

Figure 9-4 Host RMI code

Before the client can use the remote methods, the host program must bind itself as a service with an appropriate name. We use the `Naming.rebind` method to do this. It requires a URL as an input parameter. The format of the URL is:

```
// + the name of the host system + the TCP port number + the service name  
("//sysname:port/AddOne")
```

We show the host program registering with the RMI server in Figure 9-5.

```
public static void main(String[] args) {
    try{
        AddOneServer myAddOneServer = new AddOneServer();
        System.out.println("Main:Attemp to register AddOneServer");
        Naming.rebind("//AS25:6666/AddOne",myAddOneServer);
        System.out.println("Main: Successfully registered with RMI");
    }
    catch(Exception e) {
        e.printStackTrace();
    }
    return;
}
```

Figure 9-5 Registering with the RMI

### 9.3.3 Running rmic on a remote implementation class

The `rmic` command automatically creates stub and skeleton code from the interface and implementation class definitions.

To make the remote class ready to use, you must:

1. Compile server classes using `javac` or an IDE.
2. Run `rmic` or an equivalent function from an IDE.

- In the JDK, the command is: `rmic AddOneServer`
- From the VisualAge for Java IDE, right-click the **AddOneServer** class, and select **Tools-> Generate RMI-> JDK 1.2 stubs/skeletons**

The output from the `rmic` command is two new Java class files:

- ▶ `AddOneServer_skel.class`
- ▶ `AddOneServer_stub.class`

There is no need to modify these files. Some IDE tools also create `.java` files.

**Note:** VisualAge for Java will not generate a skeleton if you choose the JDK 1.2 implementation.

### 9.3.4 Implementing the client

For the client to use the methods of the remote object, it must use the `Naming.lookup` method to find the remote server object, which:

- ▶ Uses the server interface object
- ▶ Must match the port number and service name used by the server

The client can make method calls on the remote server object like any other Java method call. Figure 9-6 shows the client code.

```
public static void main(String[] args)
{
    int myNum = 0;
    try
    {
        AddOneServerInterface myAddOne = (AddOneServerInterface)Naming.
            lookup("//AS25:6666/AddOne");
        for(int i=0;i<10;i++)
        {
            System.out.println("myNum = " + (myNum = myAddOne.AddOne(myNum)));
        }
    }catch (Exception e)
    {
        e.printStackTrace();
    }
    return;
}
```

Figure 9-6 Client program

Before the client can use the remote methods, it must obtain a reference to the remote object. Use the `Naming.lookup` method to do this. It requires a URL as an input parameter. The format of the URL is:

// + the name of the host system + the TCP port number + the service name

Here is the code:

```
AddOneServerInterface myAddOne = (AddOneServerInterface)Naming.
    lookup("//AS25:6666/AddOne");
```

### 9.3.5 Making the server code network accessible

To make the server code network accessible, you must complete these steps:

1. Run the RMI Registry (`rmiregistry`) command on the server:
  - Pass in the TCP/IP port as the first parameter.
  - The CLASSPATH must provide access to all required server objects.
2. Run the server object in a new (second) server process, which binds to the registry.
3. Invoke the client.

Since we are using RMI support, we have to execute the RMI registry. The registry must run in the Qshell environment. Before starting the Qshell environment, set the Java Environment CLASSPATH information. The registry must be able to find the application that we are running. To run the RMI registry, use the following command:

```
rmiregistry 6666
```

Next, run the application. First, set the Java Environment CLASSPATH information. There are a number of ways to do this. We need to set the CLASSPATH so we can find the application class and the IBM Toolbox for Java classes (if we are using them). Then, execute the host application using this command:

```
java AddOneServer
```

We are now ready to invoke the client. It can use the remote method supplied by the host:

```
java UseAddOne
```

## 9.4 RMI JDBC example

This section explains how to implement an iSeries client/server application using RMI. This is a thin client implementation. The client handles all the graphical user interface support while all the logic and database access is performed on the server. The server uses JDBC to access the iSeries database.

Figure 9-7 on page 356 shows the main window of the RMI example. To run the example, enter the name of the iSeries server and click the Connect button. This causes the `connectToDB` method to run. If you successfully establish an RMI connection to the iSeries server, you receive the message Connected to AS/400. This program can retrieve information about one part or all of the parts in the database.

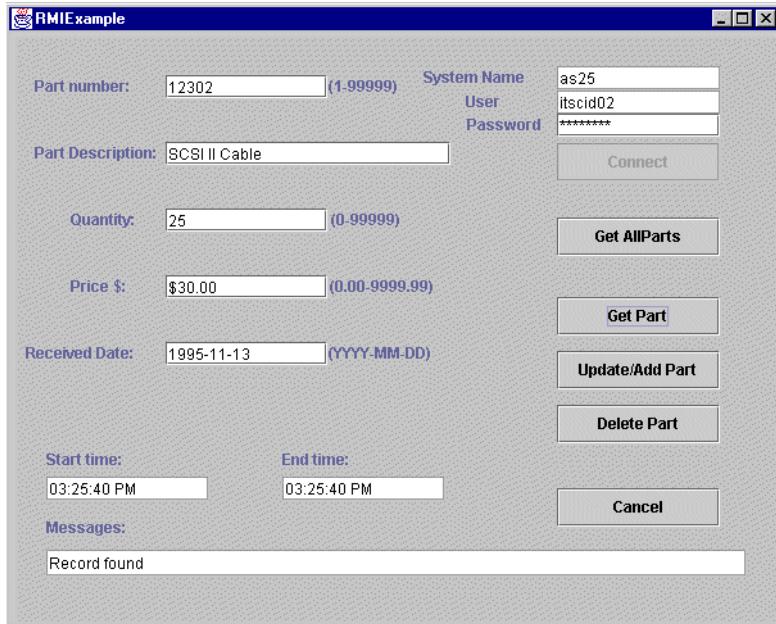


Figure 9-7 JDBC RMI application

Figure 9-8 shows the result of clicking the Get All Parts button. All part numbers are retrieved from the PART database and displayed in a table.

Part#	Description	Qty	Price \$	Received
012301	50X Speed CD ROM Drive	15	\$150.00	2001-09-09
012302	SCSI II Cable	25	\$30.00	1995-11-13
012304	Ethernet PCMCIA card	30	\$85.30	1995-12-17
012305	Home mouse	47	\$25.50	1996-02-18
012306	Gender-bender	75	\$8.50	1995-08-27
012307	600 dpi flatbed scanner	12	\$875.33	1996-03-01
012308	100 MHZ Pentium PC	4	\$1875.20	1996-02-24
012309	LaserJet Toner	12	\$89.45	1995-12-17
012310	Logo mouse mat	376	\$7.25	1994-11-24
012312	V34 Modem	60	\$120.45	1996-03-06
012313	Games joystick	32	\$42.75	1995-11-12
012314	3m printer cable	20	\$12.40	1996-01-23
012315	Anti-glare screen	45	\$34.77	1996-02-27
012316	Quad speed CD ROM Drive	14	\$151.38	1996-01-12
012317	SCSI II Cable	25	\$37.84	1995-11-13
012318	17 inch SVGA Monitor	6	\$1388.59	1996-03-04
012319	Ethernet PCMCIA card	30	\$107.60	1995-12-17
012320	Home mouse	47	\$32.16	1996-02-18
012321	Gender-bender	75	\$10.71	1995-08-27
012322	600 dpi flatbed scanner	12	\$1104.21	1996-03-01

Figure 9-8 RMI example: Getting all parts

Figure 9-9 shows the RMI example application design.

The client Java program requests data from the iSeries database by sending requests to the host Java program. The host Java program uses JDBC to access the PARTS database. Two SQL statements are used:

- ▶ Select \* from apilib.parts to retrieve all columns for all records
- ▶ Select \* from apilib.parts where partno =? to retrieve all columns for a given part number

The entire application is kept in a package named JDBCrmIExample as shown in Figure 9-10.

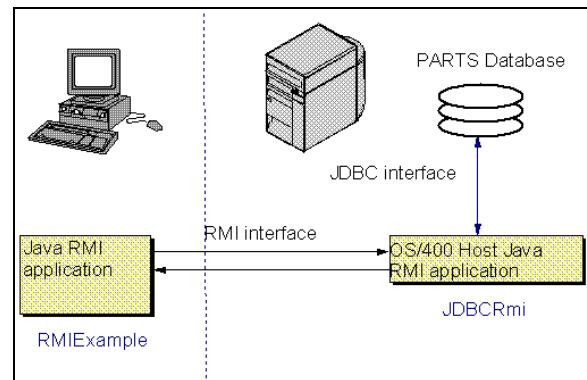


Figure 9-9 iSeries RMI example

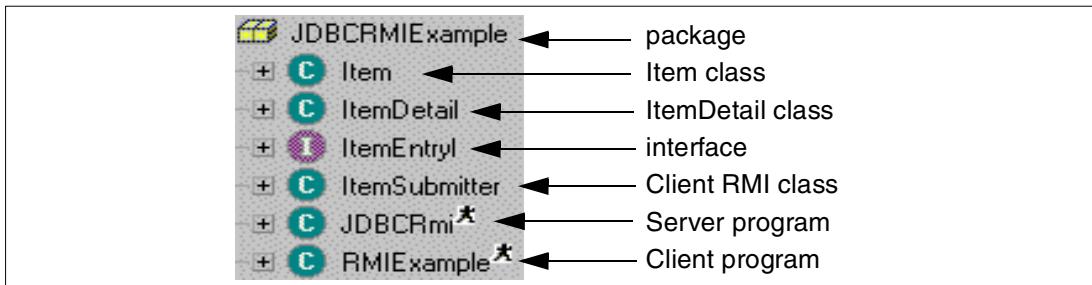


Figure 9-10 Java package for the RMI example

This package contains the following classes:

- ▶ **Item**: Used to create Item objects
- ▶ **ItemDetail**: Used to create Item Detail objects
- ▶ **ItemEntryl**: Contains the interface implemented to support RMI
- ▶ **ItemSubmitter**: Used to create an RMI support object for the client
- ▶ **JDBCrmI**: The iSeries host Java program
- ▶ **RMIEexample**: The client Java program

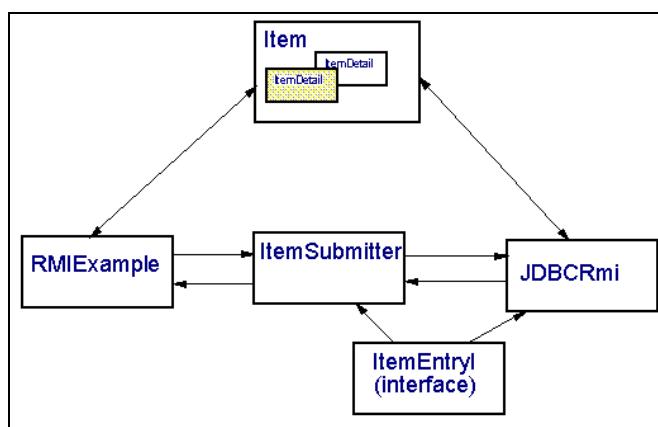


Figure 9-11 RMI application design

Figure 9-11 shows the program interface. RMIEexample is the Java client program. It creates an instance of the ItemSubmitter class. The ItemSubmitter class contains all the RMI support for the client side. ItemEntryl is the interface, which describes the public methods. It is used by ItemSubmitter and JDBCrmI (the host Java program).

The Item class is used to pass information about an item between the host java program (JDBCrmI) and the client Java program

(RMIEexample). If a request is made for all the items in the database, an item object is created that contains ItemDetail objects for each record in the database.

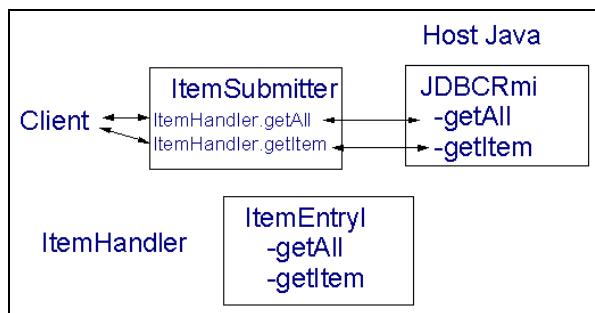


Figure 9-12 RMI example public methods

Figure 9-12 shows how the public methods are used. ItemEntryl is the interface that describes the public methods.

There are two public methods:

- ▶ public Item getAll() throws RemoteException
- ▶ public Item getItem(String anItem) throws RemoteException

The getAll() method takes no input parameters and returns an Item object. The getItem() method takes a String input parameter, which contains the part number requested, and returns an Item object, which contains the details about an Item.

To separate the RMI logic from the application logic, the ItemSubmitter class is used by the client program. The client program instantiates an instance of the ItemSubmitter class. It is called ItemHandler. The client program uses the ItemHandler object to interface to the public methods.

On the host side, JDBCRemote contains the actual application logic for the public methods:

- ▶ The getAll() method uses the JDBC interface to retrieve all records from the PARTS database.
- ▶ The getItem() method uses JDBC to retrieve the requested item from the PARTS database. In both methods, an Item object is returned to the client program.

#### 9.4.1 Item class

The Item class is used to pass information about a particular item (or part) between the client program and the server program. It contains the Item identification, Item description, Item price, Item quantity, and Item date. It also contains an array that can contain up to 100 ItemDetail objects. The ItemDetail objects are used when a request for all items (or parts) is made. In this case, one Item object is returned that contains an ItemDetail object for each item in the PARTS database. The Item class is shown in Figure 9-13.

```

import java.io.Serializable;
public class Item implements Serializable
{
    private StringBuffer ItemId = new StringBuffer(5);
    private String ItemDesc;
    private java.math.BigDecimal ItemPrice;
    private int ItemQuantity;
    private String ItemDate;
    private ItemDetail[] entryArray = new ItemDetail[100];
}

```

Figure 9-13 Item class

The Item class also provides a number of methods for accessing or changing information in the Item object(getters/setters). Methods are also provided for determining the number of ItemDetail objects contained in the Item object and for retrieving the ItemDetail objects.

The Item class implements Serializable. This is required to allow objects to be passed as parameters over a communication network.

The ItemDetail class is used to pass information about all items between the client program and the server program. It contains the Item identification, Item description, Item price, Item quantity, and the Item date. The ItemDetail objects are used when a request for all items (or parts) is made. In this case, one Item object is returned that contains an ItemDetail object for each item in the PARTS database.

The ItemDetail class also provides a number of methods for accessing information in the ItemDetail object(getters). The ItemDetail class implements Serializable. This is required to allow objects to be passed as parameters over a network. The ItemDetail class is shown in Figure 9-14.

```
public class ItemDetail implements Serializable {  
    StringBuffer itemId = new StringBuffer(6);  
    StringBuffer itemDsc = new StringBuffer(24);  
    String itemPrice;  
    String itemQty;  
    String itemDate;  
}
```

Figure 9-14 ItemDetail class

#### 9.4.2 Defining the interface

When using Java RMI support, the public methods must be described in an interface. The interface must extend the Remote class. The public methods must throw a RemoteException. The interface is named ItemEntryI and is shown in Figure 9-15.

```
import java.rmi.*;  
public interface ItemEntryI extends Remote {  
    public Item getAll() throws RemoteException;  
    public Item getItem(String anItem) throws RemoteException;  
}
```

Figure 9-15 ItemEntryI interface

#### 9.4.3 Implementing the remote server object

The code example in Figure 9-16 shows the class description for the host Java program. We import a number of support classes, including the RMI support classes. To use RMI, we must extend the UnicastRemoteObject class and implement the ItemEntryI interface.

```

import com.ibm.as400.access.*;
import java.math.*;
import java.sql.*;
import java.util.*;
import java.text.*;
import java.rmi.*;
import java.rmi.server.*;

public class JDBCRemote extends UnicastRemoteObject implements ItemEntryI
{
    // Mnemonic values
    private static final String SYSTEM = "localhost"; // update to your system name
    private static final String USER = "*current";
    private static final String PASSWORD = "*current";
    private static final String DATA_LIBRARY = "APILIB";

    // A global connection and prepared statement
    private Connection dbConnect = null;
    private PreparedStatement psAllRecord;
    private PreparedStatement psSingleRecord;
}

```

*Figure 9-16 JDBCRemote class*

The `UnicastRemoteObject` class defines the remote object as a unicast object, which means that only a single instance of the object can exist on a single server. This is distinguished from `MultiCastRemoteObject`, which can replicate across multiple servers.

The class must implement an interface that describes the public methods. In the example used here, the interface is named `ItemEntryI`. It does not need to be called `ItemEntryI`, but such a naming convention helps keep the links between the classes clear. The `Interface` class must import the `java.rmi` package to use the RMI classes.

We declare some global variables that are used in the application. The value `*current` means to use the information for the current user that is signed on. We also declare an SQL Connection object and two PreparedStatement objects that are used to access the iSeries database through the JDBC interface.

Each remote class must be capable of registering its services with an RMI registry that provides brokering services between the client and the server. We do this by adding a main method that performs the registration. The RMI classes throw exceptions so we must wrap our use of these classes in a `try{} catch{}` block. The host main method is shown in Figure 9-17.

```

public static void main(String[] parameters)
{
    // main must be invoked with 1 parameter: the port number
    // this should be the same port number on which the
    // particular Registry has been started
    if(parameters.length<1)
    {
        System.err.println("Must pass port number when invoking.");
        System.exit(1);
    }
    String port = ":"+parameters[0];
    // Set up the server
    try
    {
        JDBCRemote oeJDBC = new JDBCRemote();
        System.out.println("Main: Attempting to register JDBCRemote");
        Naming.rebind("//AS25" + port +"/JDBCRemote", oeJDBC);
        System.out.println("Main: Successfully registered with the RMI");

    } catch(Exception e) {e.printStackTrace();}

    return;
}

```

*Figure 9-17 JDBCRemote main() method*

The host initialize method shown in Figure 9-18 demonstrates how the JDBC environment is set up. It shows using the IBM Native JDBC driver to access the iSeries server database.

```

private void initialize () throws Exception
{
    // Create a properties object for JDBC connection
    Properties jdbcProperties = new Properties();
    // Set the properties for the JDBC connection
    jdbcProperties.put("user", USER);
    jdbcProperties.put("password", PASSWORD);
    jdbcProperties.put("naming", "sql");
    jdbcProperties.put("errors", "full");
    jdbcProperties.put("date format", "iso");

    Class.forName ("com.ibm.db2.jdbc.app.DB2Driver");
    // Connect using the properties object
    dbConnect = DriverManager.getConnection("jdbc:db2:"+SYSTEM+"/"+DATA_LIBRARY,
                                         jdbcProperties);
    psSingleRecord = dbConnect.prepareStatement("SELECT * FROM PARTS WHERE PARTNO = ?" );
    psAllRecord = dbConnect.prepareStatement("SELECT * FROM PARTS order by partno");
    return;
}

```

*Figure 9-18 Initializing the JDBC connection*

To connect to the iSeries database, we use the `DriverManager.getConnection()` method. The `DriverManager.getConnection()` method takes a URL string as an argument. The JDBC driver manager attempts to locate a driver that can connect to the database represented by the URL. When using the native IBM Java driver, we use the following syntax for the URL:

`jdbc:db2:systemName/defaultLibrary;listOfProperties`

An SQL statement can be compiled and stored in a PreparedStatement object. The Java program can use this object to run the statement multiple times since the statement is compiled only once. This is more efficient than running the same statement multiple times using a Statement object, which compiles the statement each time it is run. We use the Connection.prepareStatement() method to create PreparedStatement objects.

Next, we create the code for the public getItem() method. The getItem() method is shown in Figure 9-19. We first create a new Item object that we return to the caller if we successfully find the item in the database. A ResultSet object provides access to a table of data generated by running a statement.

```
public Item getItem (String anItem) throws RemoteException
{
    Item theItem = new Item(anItem);
    try
    {
        java.sql.ResultSet rs = null;
        psSingleRecord.setInt(1, Integer.parseInt(anItem));
        rs = psSingleRecord.executeQuery();
        if (rs.next()) {
            theItem.setItemDesc(rs.getString("PARTDS"));
            theItem.setItemQty(rs.getInt("PARTQY"));
            theItem.setItemDate(rs.getDate("PARTDT").toString());
            theItem.setItemPrice(rs.getBigDecimal("PARTPR").setScale(2));
        }
        else {
            return(null);
        }
    } catch (Exception e) {e.printStackTrace(); return null; }

    return(theItem);
}
```

Figure 9-19 The getItem() method of the remote object

The table rows are retrieved in sequence. Within a row, column values can be accessed in any order:

- ▶ **java.sql.ResultSet rs = null;**  
Declares a variable, rs, to reference a ResultSet object.
- ▶ **psSingleRecord.setInt(1, Integer.parseInt(partNo));**  
Uses the setInt() method of PreparedStatement to set the value of first parameter to the integer value of the part number passed on the parameter list.
- ▶ **rs = psSingleRecord.executeQuery();**  
Executes the SQL defined by the psSingleRecord object and places the table of resulting records in a ResultSet object referenced by rs.
- ▶ **if (rs.next())**  
The next() method of the ResultSet attempts to position the cursor of the result set to the next record from the result table. Since this is the first method read from the ResultSet, the method positions to the first record from the ResultSet and returns true. If there are no records to retrieve, the method returns a false value.

The following lines retrieve values of database fields and place them in the Item object that is returned:

- theItem.setItemDesc(rs.getString("PARTDS"));
- theItem.setItemQty(rs.getInt("PARTQY"));
- theItem.setItemDate(rs.getDate("PARTDT").toString());
- theItem.setItemPrice(rs.getBigDecimal("PARTPR").setScale(2));

We use the setter provided by the Item class to do this. ResultSet objects have getter methods for many Java data types.

Here we use:

- **getString()**: Returns the value of the column PARTDS as a String object
- **getInt()**: Returns the value of the column PARTQY as an integer
- **getBigDecimal()**: Returns the value of the PARTPR field as a BigDecimal object
- **getDate()**: Returns the value of column PARTDT as a Date

If we successfully find the item number in the database, we return the Item object. Otherwise, we return null. Next, we write the code for the getAll() method.

The getAll() method is shown in Figure 9-20. To get all the records, we execute the Prepared Statement object named psAllRecord. It does not require any parameters. In this case, we have multiple rows returned in the result set. We use the result set, Next method, to retrieve each row from the result set. For each row returned, we create an ItemDetail object in the Item object that we return to the caller.

```
public Item getAll () throws RemoteException
{
    Item theItem = new Item("all");
    java.sql.ResultSet rs = null;
    String[] detailRow = new String[5];
    try {
        rs = psAllRecord.executeQuery();
        while (rs.next()) {
            ItemDetail detail = new ItemDetail(
                rs.getString("PARTNO"),
                rs.getString("PARTDS"),
                Integer.toString(rs.getInt("PARTQY")),
                "$" + rs.getBigDecimal("PARTPR").setScale(2).toString(),
                rs.getDate("PARTDT").toString());

            theItem.addEntry(detail);
        }
    } catch(Exception e) {e.printStackTrace(); return null;}
    return(theItem);
}
```

Figure 9-20 The getAll() method of remote object

#### 9.4.4 Creating the stubs and skeletons

To make the remote class ready to use, you must complete this process:

1. Compile server classes using the **javac** command or an IDE.
2. Run **rmic** or its equivalent function from an IDE.
  - In the JDK, the command is:

```
rmic JDBCRemote
```

- From VisualAge for Java, follow this sequence **Tools-> Generate RMI-> JDK 1.2 stubs/skeletons** (as shown in Figure 9-21).

The output from the `rmic` command is two new Java class files:

- **JDBCRmi\_skel.class**
- **JDBCRmi\_stub.class**

There is no need to modify these files. Some IDE tools also create .java files.

**Note:** VisualAge for Java will not generate a skeleton if you choose the JDK 1.2 implementation.

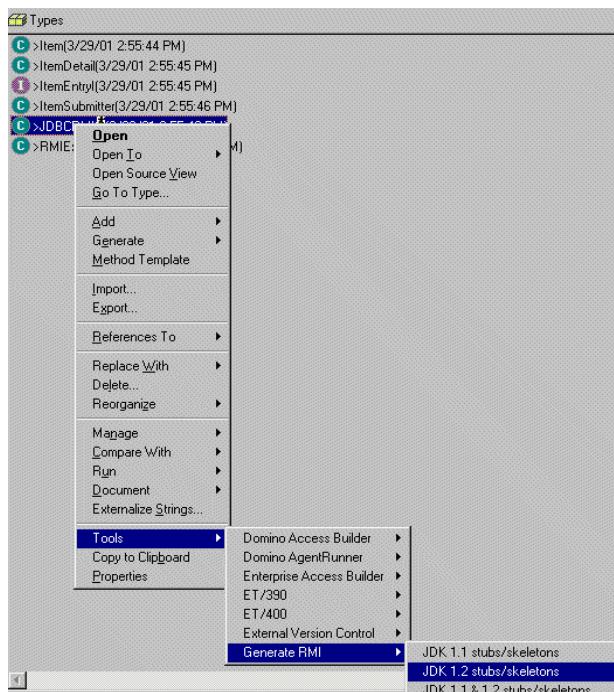


Figure 9-21 Creating the stubs and skeletons in VisualAge for Java

After running Generate RMI, our package now contains all of the required host remote classes as shown in Figure 9-22.

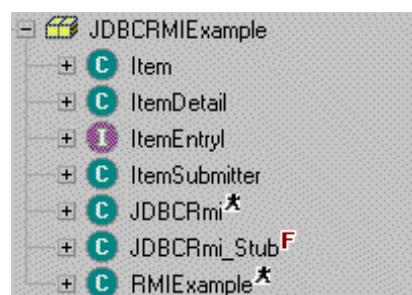


Figure 9-22 Completed host remote application

Before we can run the application on the iSeries server, we have to move it there. If the iSeries IFS is available as a network drive, we can export the host classes to the iSeries server.

We directly export the classes using the VisualAge for Java export function. When we run the application on the iSeries server, we have to set the CLASSPATH environment variable to include the IFS file directory where we stored the class files. In Figure 9-23, we export the classes to the iSeries server using a network drive.

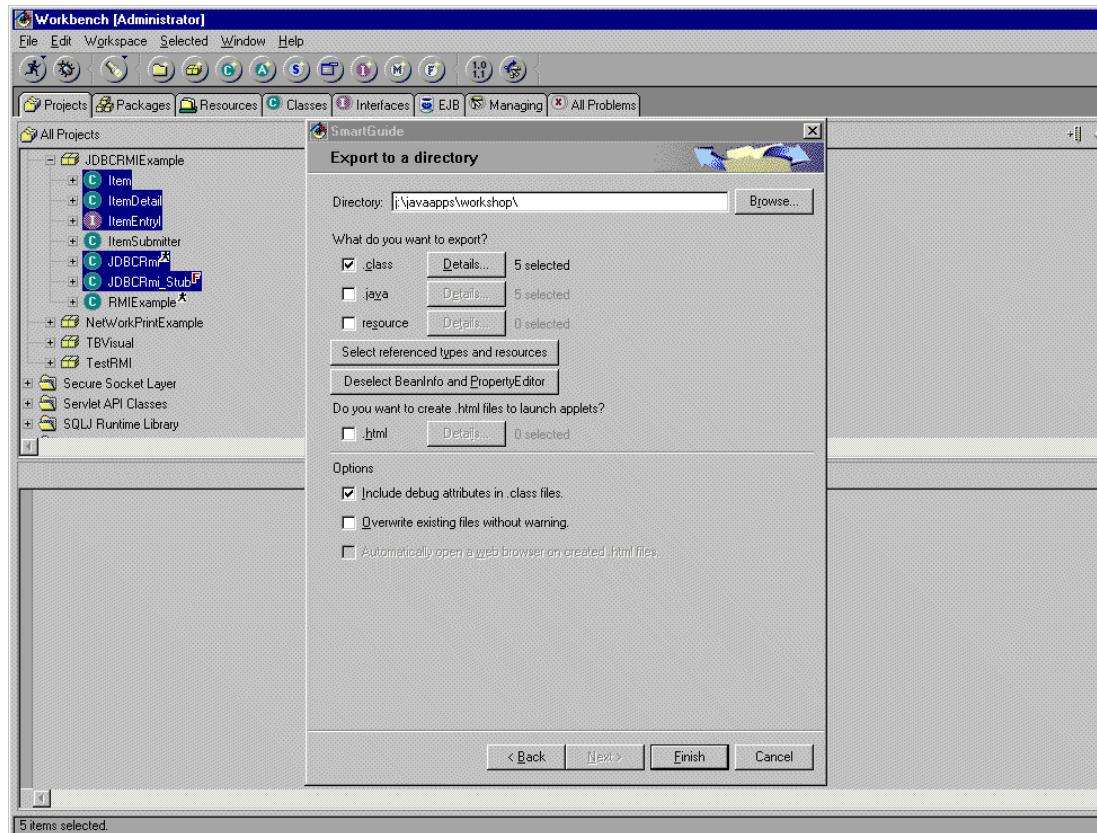


Figure 9-23 Exporting the class files

#### 9.4.5 Implementing the client

This section explains how to build the client class of the RMI application. We create a new class that is a subclass of javax.swing.JFrame and implements the PartsContainer interface. This class uses the ToolboxGUI class as re-usable part. For details about the ToolboxGUI class and the PartsContainer interface, refer to 4.6.5, “Reusable GUI part” on page 170.

The name of the client program is RMIExample. As shown in Figure 9-24, we create global variables for the port name and system name and create a new instance of the ItemSubmitter class, which we call remoteRequestor. We use the remoteRequestor object for all of our RMI work.

```
public class RMIExample extends javax.swing.JFrame implements
DatabaseAccessExample.PartsContainer, java.awt.event.WindowListener
{
    private DatabaseAccessExample.ToolboxGUI ivjToolboxGUI1 = null;
    static java.lang.String port = null;
    static java.lang.String systemName;
    ItemSubmitter remoteRequestor = new ItemSubmitter();
```

Figure 9-24 Creating the client class

The RMIEexample main method (Figure 9-25) was created by VisualAge for Java.

```
public static void main(java.lang.String[] args) {
    if ((port = args[0]) != null)
    {
        try {
            RMIEexample aRMIEexample;
            aRMIEexample = new RMIEexample();
            aRMIEexample.addWindowListener(new java.awt.event.WindowAdapter() {
                public void windowClosing(java.awt.event.WindowEvent e) {
                    System.exit(0);
                }
            });
            aRMIEexample.show();
            java.awt.Insets insets = aRMIEexample.getInsets();
            aRMIEexample.setSize(aRMIEexample.getWidth() + insets.left + insets.right,
                                 aRMIEexample.getHeight() + insets.top + insets.bottom);
            aRMIEexample.setVisible(true);
        } catch (Throwable exception) {
            System.err.println("Exception occurred in main() of javax.swing.JFrame");
            exception.printStackTrace(System.out);
        }
    }
    else
    {
        System.out.println("Please Input port number");
    }
}
```

Figure 9-25 RMIEexample main() method

When the application is run, the `main()` method instantiates a new instance of the `RMIEexample` class. We only have to modify it to use the first argument of the parameter list to set the `port` variable.

The `connectToDB()` method shown in Figure 9-26 is executed when the user clicks the Connect button.

```
public void connectToDB(String systemName, String userid, String password) throws
Exception {
    this.systemName = systemName;
    remoteRequestor.linked(systemName, port);
    return;
}
```

Figure 9-26 RMIEexample connectToDB() method

We use the system name from the screen `TextField` as a parameter to set the `systemName` variable. We call the `linked()` method of the `remoteRequestor` object, which was instantiated from the `ItemSubmitter` class to establish the RMI connection.

The `getRecord()` method, shown in Figure 9-27, is called when the user clicks the Get Part button.

```

public String getRecord(String partNo, javax.swing.JTextField partDesc,
    javax.swing.JTextField partQty, javax.swing.JTextField partPrice,
    javax.swing.JTextField partDate) throws Exception {

    Item rtnItem = remoteRequestor.submit(partNo);
    if ((rtnItem) != null)
    {
        partDesc.setText(rtnItem.getItemDesc());
        partDate.setText(rtnItem.getItemDate());
        partQty.setText((Integer.toString(rtnItem.getItemQuantity())));
        partPrice.setText("$" + (rtnItem.getItemPrice()).toString());
    }
    else {
        partDesc.setText("");
        partDate.setText("");
        partQty.setText("");
        partPrice.setText("");
        return "Record not found";
    }
    return "Record found";
}

```

*Figure 9-27 RMIExample getRecord() method*

The `getRecord()` method calls the `submit()` method of the `remoteRequestor` object passing the part number as a parameter. An `Item` object is returned that contains the details about the item (or Part).

If an `Item` object is returned, the item requested was found in the database. We use the getter methods provided by the `Item` object to retrieve the information and display it on the screen.

The `populateAllParts()` method, shown in Figure 9-28, is called when the user clicks the Get All Parts button. It calls the `submitAll()` method of the `remoteRequestor` object passing no parameters.

```

public void populateAllParts(javax.swing.table.DefaultTableModel defaultTableModel)
throws Exception
{
    Item rtnItem = remoteRequestor.submitAll();
    if (rtnItem != null) {
        for (int i = 0; i < rtnItem.getNumEntries(); i++) {
            ItemDetail rtnDetail = rtnItem.getNextEntry(i);
            String[] array = new String[5];
            array[0] = rtnDetail.getItemId();
            array[1] = rtnDetail.getItemDsc();
            array[2] = rtnDetail.getItemPrice();
            array[3] = rtnDetail.getItemQty();
            array[4] = rtnDetail.getItemDate();
            defaultTableModel.addRow(array);
        }
    }
    return;
}

```

*Figure 9-28 RMIExample populateAllParts() method*

An Item object is returned that contains an array of ItemDetail objects, which contain the details about each item (or Part) in the database. If an Item object is returned, we use the getNumEntries() method to determine how many ItemDetail objects were returned. We use the getter methods provided by the ItemDetail object to retrieve the information and add it to the array, which is used to populate the DefaultTableModel object.

#### 9.4.6 Making the server code network accessible

Since we are using RMI support, we have to run the RMI registry. The registry must run in the Qshell environment. Before starting the Qshell environment, we set the Java environment CLASSPATH information. There are a number of ways to do this. Here, we use the Add Environment Variable (ADDENVVAR) command. The registry must be able to find the application that we are running. We run the RMI registry using the **rmiregistry** command. Setting the CLASSPATH environment variable and starting the Qshell environment is shown in Figure 9-29.

The screenshot shows a Qshell session window. At the top, it says "Command Entry" and "AS25 Request level: 4". Below that, it displays previous commands and messages, including the execution of the ADDENVVAR command to set the CLASSPATH environment variable to 'javaapps/workshop'. It also shows the prompt "Type command, press Enter." followed by "====> qsh". At the bottom of the window, there is a legend for function keys: F3=Exit, F4=Prompt, F9=Retrieve, F10=Include detailed messages, F11=Display full, F12=Cancel, F13=Information Assistant, and F24=More keys.

Figure 9-29 Setting the CLASSPATH environment variable

In Figure 9-30, we run the RMI registry using the **rmiregistry** command. We pass in the TCP port number as a parameter. If port number is not specified, then default port 1099 will be used. The **rmiregistry** command does not support the verbose parameter. In this case, we run the RMI registry using a 5250 emulation session. We can also submit this as a batch job and not tie up a 5250 session.

## QSH Command Entry

*Figure 9-30 Executing the RMI registry*

Next, we run the host application. As shown in Figure 9-31, we first set the Java environment CLASSPATH information. There are a number of ways to do this. Here, we use the ADDENVVAR command again. We must set the CLASSPATH so we can find the application class and, if we are using them, the IBM Toolbox for Java classes. Then, we use the `java` CL command to execute the application passing in the port number as a parameter.

Run Java Program (JAVA)

Type choices, press Enter.

Class . . . . . JDBCrmIExample.JDBCrmI

Parameters . . . . . 8888

+ for more values

Classpath . . . . . \*ENVVAR

Bottom

F3=Exit F4=Prompt F5=Refresh F10=Additional parameters F12=Cancel  
F13=How to use this display F24=More keys

*Figure 9-31 Executing the host application*

Figure 9-32 on page 370 shows the host application screen after a successful start. The messages are output by the application writing to a standard out using `System.out.println`.

```

Java Shell Display

initialize:new
Main: Attempting to register JDBCRemote
Main: Successfully registered with the RMI

====>

F3=Exit F6=Print F9=Retrieve F12=Exit
F13=Clear F17=Top F18=Bottom F21=CL command entry

```

*Figure 9-32 Host application successful start*

We are now ready to run the client application. Both the registry and the host application are started.

After executing the client application, perform the following tasks:

1. Enter the iSeries system name, user ID, and password. Click **Connect**.
2. After the Connected to AS/400 message appears, complete the following steps:
  - a. Enter a part (item) number and click the **Get Part** button. Valid numbers are 12301 through 12350.
  - b. Click the **Get All Parts** button to display all the records in a table.

## 9.5 Moving the Order Entry server application to Java

So far, we have shown you how to move the user interface for the Order Entry application to a client PC using Java. We have demonstrated how to do this while reusing much of the existing RPG application. Now, we are ready to replace the RPG application with a Java version. The primary benefit of this is to gain easier maintenance and portability of our application.

Next, we migrate the Order Entry application to Java on iSeries. We use the Java remote method invocation interface to allow the client Java program to interface with the server Java code. Figure 9-33 shows the new design.

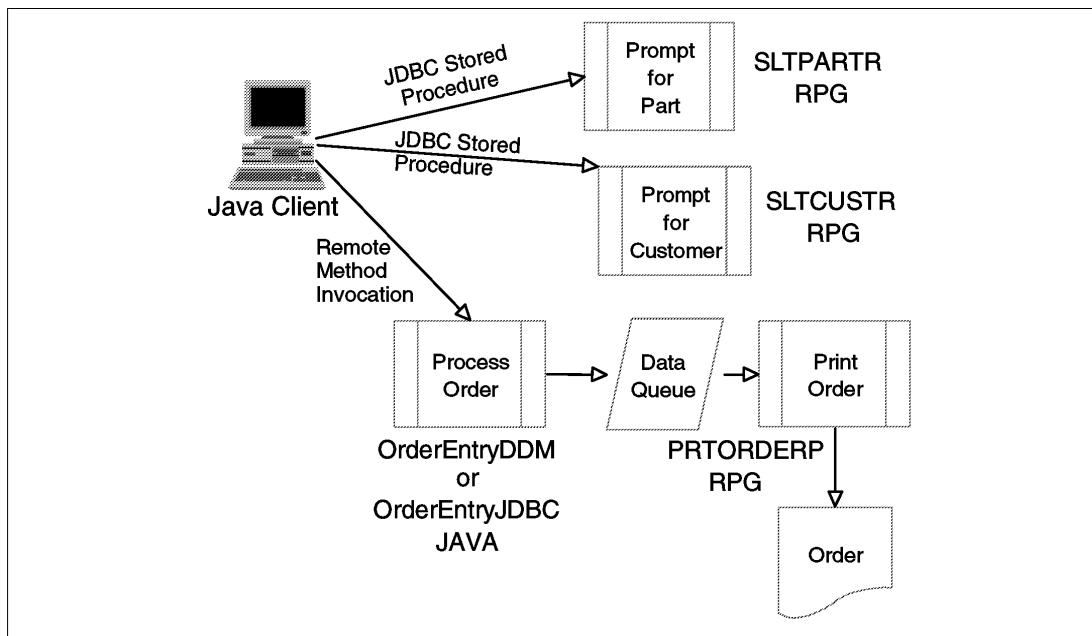


Figure 9-33 Java client/Java server Order Entry application

This section explains the Java code that is necessary to replace the RPG Order Entry application. We discuss two techniques: how to use record-level access and how to use JDBC. We also discuss the changes that are required for the client code.

There are three approaches to creating Java on iSeries for the Order Entry application:

- ▶ We can simply create a procedural Java program and make all variables and methods public. This is the most straightforward way of moving to Java, but it does not take advantage of any of the object-oriented constructs, such as encapsulation.
- ▶ We can completely redesign the application to be fully object-oriented. This involves creating classes to encapsulate all of the files that are used, a class to hide the data queue implementation, and classes to describe an order and a customer.
- ▶ We can compromise and use object-oriented constructs where it seems sensible to do so and still use a somewhat procedural coding style for the primary methods.

We have chosen to take the third approach, because it seems to be easier to understand as a first step in moving to Java. We use classes to describe the Order Entry application and the order itself. We also hide the internal implementation of the Order Entry class.

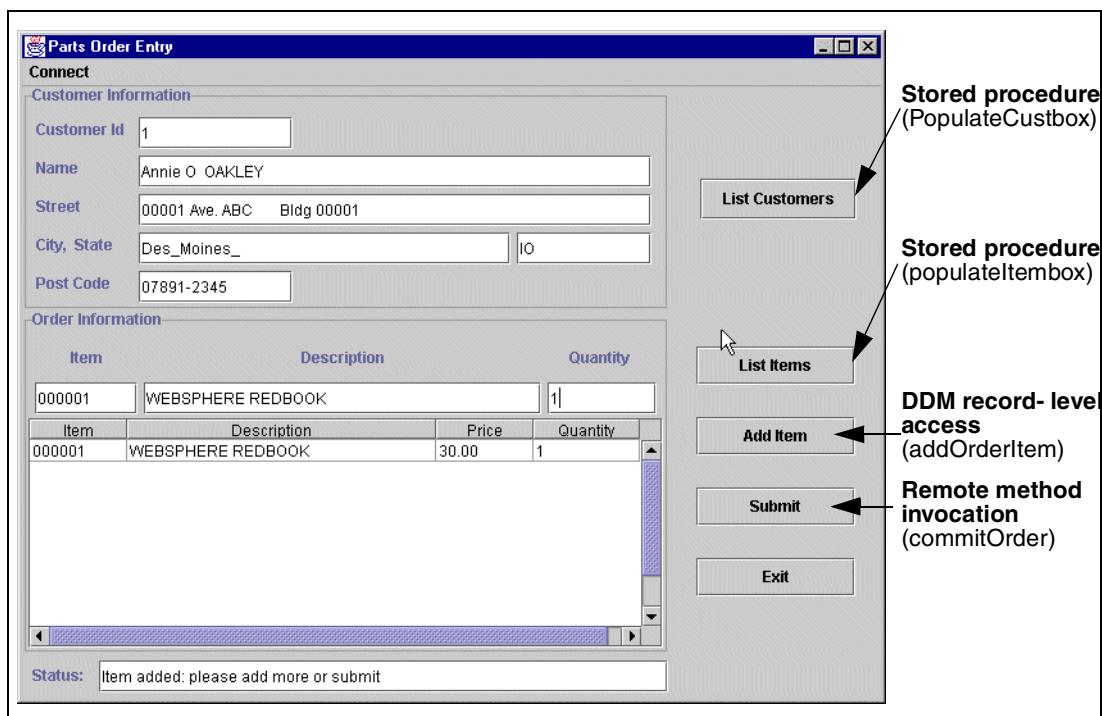


Figure 9-34 Java client programming interfaces

The client Java program is the Java client program that was covered in Chapter 8, “Migrating the user interface to the Java client” on page 321. The difference is that now, the submit order processing is done through the Java RMI interface. The following sections cover implementing the RMI interface for this application.

### **Mapping RPG to Java**

The first step in converting our RPG application to Java is to create a class that represents the RPG program. Then, we map the RPG subroutines to Java methods. The last step is to write the code to implement the methods.

Creating a single class to represent an Order Entry program is the simplest design for a Java replacement. We need to consider which techniques to use to access the iSeries database. These are the four choices:

- ▶ JDBC-ODBC bridge
- ▶ IBM Toolbox for Java JDBC running on the iSeries server
- ▶ IBM Toolbox for Java Record Level Access (DDM) running on the iSeries server
- ▶ IBM Developer Kit for Java JDBC (Native JDBC)

We demonstrate two different techniques of implementing the Java server application. They are DDM record-level access and Native JDBC. The two implementations are functionally equivalent and are both equivalent to the RPG application that was discussed in Chapter 6, “Overview of the Order Entry application” on page 299.

We create two classes. The OrderEntryDDM class is used for the DDM record-level access example. The OrderEntryJDBC class is used for the JDBC example. Each class contains Java methods that map to the subroutines that are found in the RPG example. We initially implement record-level access, because it is closer to the native I/O mechanisms that were used in the original RPG application and, therefore, are easier to understand.

## 9.6 Order Entry using record-level access (DDM)

In this section, we create the OrderEntryDDM class and its supporting methods. Complete the following steps:

1. Create a class called OrderEntryDDM in the serverOrderEntry package. The class and its methods are shown in Figure 9-35.

```
serverOrderEntry Package
    OrderEntryDDM Class
        commitOrder()
        addOrderLine()
        addOrderHeader()
        updateStock()
        updateCustomer()
        writeDataQueue()
        getOrderNumber()
        finalize()
        initialize()
```

Figure 9-35 OrderEntryDDM class

2. Specify that the class OrderEntryDDM is in the serverOrderEntry package:

```
package serverOrderEntry;
```

3. Define the packages that are used by this class:

```
import com.ibm.as400.access.*; // for IBM Toolbox for JavaIBM Toolbox for Java classes
import java.math.*; // for BigDecimal class
import java.text.*; // for DateFormat class
import java.util.*; // for Properties class
```

This is the class definition. The class can be used by any other object.

```
/**
 * This class is a replacement for the ORDENTR RPG IV program.
 * The method and variable names have been improved slightly
 * since Java supports longer names than RPG.
 */
public class OrderEntryDDM
{
```

4. Define some named constants to simplify code changes in the methods of the class. The values for *your-library*, *your-user-id*, and *your-password* need to be set appropriately. You can use the value *\*current* for the user ID and password. In this case, the user ID and password for the current iSeries session are used.

```
// Mnemonic values
private static final String SYSTEM_LIBRARY = "QSYS.LIB";
private static final String DATA_QUEUE_NAME = "ORDERS.DTAQ";
private static final String DATA_QUEUE_LIBRARY = "your-library.LIB";
private static final String WAREHOUSE = "0001";
private static final int DISTRICT = 1;
private static final String SYSTEM = "localhost";
private static final String USER = "your-user-id";
private static final String PASSWORD = "your-password";
private static final String DATA_LIBRARY = "your-library";
```

5. Create some global instance variables that are visible to all methods of the class and are global to make referencing these objects easier:

```

// an AS400 object
private AS400 as400 = null;

// File objects
private SequentialFile ordersFile = null;
private SequentialFile orderLineFile = null;
private KeyedFile customerFile = null;
private KeyedFile stockFile = null;
private KeyedFile itemFile = null;
private KeyedFile districtFile = null;

// Record formats
private RecordFormat ordersFormat = null;
private RecordFormat orderLineFormat = null;
private RecordFormat customerFormat = null;
private RecordFormat stockFormat = null;
private RecordFormat itemFormat = null;
private RecordFormat districtFormat = null;

```

6. Define the default constructor for the class. The default constructor is responsible for initializing the object when a new instance is created. It ensures that its super class is initialized and then performs its own initialization.

```

/**
 * This method was created by a SmartGuide.
 */
public OrderEntryDDM () throws Exception
{
    super();
    initialize();
}

```

The basic structure of the OrderEntryDDM class has now been completed. At this point, we have the ability to create an instance of the class. However, it cannot do anything until we define and implement the methods that the class provides. There is almost a one-to-one relationship between the RPG subroutines and the methods provided by this class.

- ▶ **CmtOrder2:** commitOrder()
- ▶ **AddOrdLin:** addOrderLine()
- ▶ **AddOrdHdr:** addOrderHeader()
- ▶ **UpdStock:** updateStock()
- ▶ **UpdCust:** updateCustomer()
- ▶ **WrtDtaQ:** writeDataQueue()
- ▶ **GetOrdNbr:** getOrderNumber()
- ▶ **EndPgm:** finalize()
- ▶ **\*INZSR:** initialize()

Next, we add these methods to the class definition. The methods are inserted after the default constructor.

**Note:** Most of the methods are defined as *private* to hide the internal implementation of the OrderEntryDDM class from the users of the class. The only external interface to the OrderEntryDDM class is the constructor and the commitOrder() method.

### ***The initialize() method***

This is the basic initialize() method. It is responsible for ensuring that the new object has the correct starting values.

```

private void initialize ()
{
    return;
}

```

### ***The commitOrder() method***

This is the basic commitOrder() method. It is the public interface to the OrderEntryDDM class. It is responsible for accepting an Order object and processing it. It needs to know the order and it returns a string that indicates whether the order was processed successfully.

```

public String commitOrder (Order anOrder)
{
    return("Order processed successfully.");
}

```

### ***The addOrderHeader() method***

This is the basic addOrderHeader() method. It is responsible for inserting a new record in the ORDERS file. It needs to know which customer the order is for, the order identifier, and how many lines of items are in the order.

```

private void addOrderHeader (String aCustomerNbr,
                           BigDecimal anOrderNbr,
                           BigDecimal anOrderLineCount)
{
    return;
}

```

### ***The addOrderLine() method***

This is the basic addOrderLine() method. It is responsible for inserting a record in the ORDLIN file for each order line that is created by the Order Entry application. It needs to know the order identifier and the actual order.

**Note:** The Order is another object. This method returns the total value of the order to its caller.

```

private BigDecimal addOrderLine (BigDecimal anOrderNbr,
                               Order anOrder )
{
    return orderTotal;
}

```

### ***The getCustomerDiscount() method***

This is the basic getCustomerDiscount() method. It is responsible for determining the amount of discount to which a particular customer is entitled. It needs to know the customer identifier and it returns the amount of that discount to its caller.

```

private BigDecimal getCustomerDiscount (String aCustomerID)
{
    return customerDiscount;
}

```

### ***The getOrderNumber() method***

This is the basic getOrderNumber() method. It is responsible for obtaining the correct identifier for this order. It simply returns an order number.

```

private BigDecimal getOrderNumber ()
{
    return orderNumber;
}

```

### **The updateCustomer() method**

This is the basic updateCustomer() method. It is responsible for updating the customers record in the CSTMR file to show the current amount ordered and the date and time of the most recent order. It needs to know the customer identifier and the value of the current order.

```
private BigDecimal updateCustomer (String aCustomerID,
                                BigDecimal anOrderTotal )
{
    return;
}
```

### **The updateStock() method**

This is the basic updateStock() method. It is responsible for ensuring that the stock quantity is reduced by the number of items ordered. It needs to know which part was ordered and how many were ordered.

```
private void updateStock (String aPartNbr,
                        BigDecimal aPartQty )
{
    return;
}
```

### **The writeDataQueue() method**

This is the basic writeDataQueue() method. It is responsible for sending a message to the Orders data queue to initiate printing of the order. It needs to know the customer identifier and the order identifier.

```
private void writeDataQueue (String aCustomerID,
                            BigDecimal anOrderID )
{
    return;
}
```

## **9.6.1 Method logic**

Next, we add the program logic to each of the methods that we created. We also need to consider exception handling. Many of the classes that we use to access the iSeries database send an error message if they encounter problems. The mechanism that Java uses to perform this is called *throwing an exception*. This is similar to the exception handling model of the iSeries server.

Consider a CL command, such as the Display Object Description (DSPOBJD) command. This command sends an \*ESCAPE message if you try to display the description of an object that does not exist. That message is known as an *exception*. If you use the DSPOBJD command in a CL program, you need to monitor for the exception if you want your program to continue running if an exception occurs.

CL monitors for exceptions with the Monitor Message (MONMSG) command. Java monitors for an exception with the try{} and catch{} blocks. You can use the MONMSG command globally in Java by adding the throws Exception statement to the definition of a method.

### **The initialize() method**

This is the complete initialize() method. This method is run by the constructor for the OrderEntry class. First, we create a connection to the iSeries server (the values SYSTEM, USER, and PASSWORD are named constants found in the class definition). This processes the record-level access requests.

**Note:** It is necessary to provide a user ID and password even when running on the same iSeries server as the database to which you are connected. You can use the value \*current for the user ID and password. In this case, the user ID and password for the current iSeries session is used.

The next block of code creates objects that represent the files that are used by the OrderEntry class. We choose an object that is appropriate to the type of processing that is performed. The ORDERS file and ORDLIN file are only written, so they can be processed sequentially. We use instances of the SequentialFile class to represent these files. All of the other files are processed randomly, so support for keyed reads is required. We use instances of the KeyedFile to represent these files. Here is an example:

```
ordersFile = new SequentialFile(as400,  
    "/QSYS.LIB/"+DATA_LIBRARY+.LIB/ORDERS.FILE/%FILE%.MBR");
```

This single line of Java code sets the ordersFile variable to reference a new SequentialFile using the AS400 connection object and the name of the database file.

Then, we create objects to represent a file description. We need these objects to retrieve the record formats for each file. Here is an example:

```
AS400 systemFileRecordDescription ordersFileD = new  
AS400FileRecordDescription(as400,  
    "/QSYS.LIB/"+DATA_LIBRARY+.LIB/ORDERS.FILE");
```

This single line of Java code declares a variable called ordersFileD, which is a type of AS400FileRecordDescription. Then, the code initializes the variable to reference a new instance of an AS400FileRecordDescription that represents a description of the file named "/QSYS.LIB/"+DATA\_LIBRARY+.LIB/ORDERS.FILE" on the server that is represented by the as400 connection.

**Note:** We are using the integrated file system naming convention even though we are connecting to the DB2 UDB for iSeries system database.

Next, we create objects to represent the record format for each file. These are used in the same way that a record format is used in an iSeries server high-level language. The record format describes the data for read, update, and write operations. The record format is retrieved by an instance method of the file description objects just created. Here is an example:

```
RecordFormat ordersFormat = ordersFileD.retrieveRecordFormat()[0];
```

This line of Java code defines a variable named ordersFormat, which is a type of RecordFormat. The code initializes the variable to the first (and in this case, only) record format of the ordersFileD file description. Array subscripts in Java start at zero.

Then, we define the key fields for the files that are processed randomly. The retrieveRecordFormat() method sets the key values automatically if the file has a primary key defined. However, we explicitly define the key to provide an example of how this is done.

The final task is to associate each record format with the corresponding file. This is similar to the way RPG associates I-specs describing the file layout with the F-specs describing the file. This happens even for externally described files.

Any exceptions that the methods generate are simply passed back to the caller.

```

private void initialize () throws Exception
{
    // Create an AS400 system connection object
    as400 = new AS400(SYSTEM, USER, PASSWORD);

    // Create the various file objects
    ordersFile = new SequentialFile(as400,
        "/QSYS.LIB/"+DATA_LIBRARY+.LIB/ORDERS.FILE/%FILE%.MBR");
    orderLineFile = new SequentialFile(as400,
        "/QSYS.LIB/"+DATA_LIBRARY+.LIB/ORDLIN.FILE/%FILE%.MBR");
    customerFile = new KeyedFile(as400,
        "/QSYS.LIB/"+DATA_LIBRARY+.LIB/CSTMR.FILE/%FILE%.MBR");
    stockFile = new KeyedFile(as400,
        "/QSYS.LIB/"+DATA_LIBRARY+.LIB/STOCK.FILE/%FILE%.MBR");
    itemFile = new KeyedFile(as400,
        "/QSYS.LIB/"+DATA_LIBRARY+.LIB/ITEM.FILE/%FILE%.MBR");
    districtFile = new KeyedFile(as400,
        "/QSYS.LIB/"+DATA_LIBRARY+.LIB/DSTRCT.FILE/%FILE%.MBR");

    // Create record description objects
    AS400 FileRecordDescription ordersFileD = new
        AS400FileRecordDescription(as400,
            "/QSYS.LIB/"+DATA_LIBRARY+.LIB/ORDERS.FILE");
    AS400 FileRecordDescription orderLineFileD = new
        AS400FileRecordDescription(as400,
            "/QSYS.LIB/"+DATA_LIBRARY+.LIB/ORDLIN.FILE");
    AS400 FileRecordDescription customerFileD = new
        AS400FileRecordDescription(as400,
            "/QSYS.LIB/"+DATA_LIBRARY+.LIB/CSTMR.FILE");
    AS400 FileRecordDescription stockFileD = new
        AS400FileRecordDescription(as400,
            "/QSYS.LIB/"+DATA_LIBRARY+.LIB/STOCK.FILE");
    AS400 FileRecordDescription itemFileD = new
        AS400FileRecordDescription(as400,
            "/QSYS.LIB/"+DATA_LIBRARY+.LIB/ITEM.FILE");
    AS400 FileRecordDescription districtFileD = new
        AS400FileRecordDescription(as400,
            "/QSYS.LIB/"+DATA_LIBRARY+.LIB/DSTRCT.FILE");

    // Get the external description of the file objects
    ordersFormat = ordersFileD.retrieveRecordFormat()[0];
    orderLineFormat = orderLineFileD.retrieveRecordFormat()[0];
    customerFormat = customerFileD.retrieveRecordFormat()[0];
    stockFormat = stockFileD.retrieveRecordFormat()[0];
    itemFormat = itemFileD.retrieveRecordFormat()[0];
    districtFormat = districtFileD.retrieveRecordFormat()[0];

    // Define the key fields for the record formats
    // The retrieveRecordFormat() method will create the default key
    //this code is to show you how to do it if you need to define a key
    customerFormat.addKeyFieldDescription("CID");
    customerFormat.addKeyFieldDescription("CDID");
    customerFormat.addKeyFieldDescription("CWID");

    stockFormat.addKeyFieldDescription("STWID");
    stockFormat.addKeyFieldDescription("STIID");
    itemFormat.addKeyFieldDescription("IID");
    districtFormat.addKeyFieldDescription("DID");
    districtFormat.addKeyFieldDescription("DWID");
}

```

```

    // Associate the record format objects with the file objects
    ordersFile.setRecordFormat(ordersFormat);
    orderLineFile.setRecordFormat(orderLineFormat);
    customerFile.setRecordFormat(customerFormat);
    stockFile.setRecordFormat(stockFormat);

    itemFile.setRecordFormat(itemFormat);
    districtFile.setRecordFormat(districtFormat);

    return;
}

```

### The commitOrder() method

This is the complete `commitOrder()` method. This method is the public interface to the `OrderEntry` class. It needs to have an order to process. The `Order` class is described in Chapter 8, “Migrating the user interface to the Java client” on page 321. It contains this logic:

- ▶ Request the customer number and the number of order lines from the order object.
- ▶ Determine the next order number by calling the `getOrderNumber()` method.
- ▶ Pass the order and the order number to the `addOrderLine()` method that adds the line items to the database.
- ▶ Add an order header record and update the customer record with information about the current order.

If an error occurred during the processing, we indicate failure by returning an error message to the caller. If processing was successful, we return a successful completion message.

We use a `try{}` and `catch{}` block to determine whether processing was successful. The code attempts to run the block of code and catches any exception. If an exception occurs, it prints a trace of the method and class stack to the Java console. Then, it returns an error message to the caller.

```

public String commitOrder (Order anOrder) throws RemoteException
{
    try
    {
        // Extract the customer number and count of lines
        String customerNumber = anOrder.getCustomerId();
        BigDecimal orderLineCount = new BigDecimal(anOrder.getOrderDetail().size());

        // Determine the order number
        BigDecimal orderNumber = getOrderNumber();

        // Add the line items to the order detail file
        BigDecimal orderTotal = addOrderLine(orderNumber, anOrder);

        // Add the order header
        addOrderHeader(customerNumber, orderNumber, orderLineCount);

        // Update the customer
        updateCustomer(customerNumber, orderTotal);

        // Commit the database changes
        // If any file is opened under commitment control we need to perform a commit
        if (ordersFile.isCommitmentControlStarted() |||
            orderLineFile.isCommitmentControlStarted() ||
            customerFile.isCommitmentControlStarted() ||
            stockFile.isCommitmentControlStarted())
    }
}

```

```

        districtFile.isCommitmentControlStarted()  )
{
    // A commit for any file under commitment control will affect all files
    customerFile.commit();
}

// Initiate order printing
writeDataQueue(customerNumber, orderNumber);

} catch(Exception e) {e.printStackTrace(); return("Order processing failed.");}

return("Order processed successfully.");
}

```

### The addOrderHeader() method

This is the complete addOrderHeader() method. It determines the current date by creating a Date object. The constructor must be qualified, because there are two Date classes available: one in *java.util* and another in *java.sql*. Then, it opens the ORDERS file and creates an empty order record. Next, the fields in the order record are populated. Finally, the record is written to the ORDERS file.

**Note:** The data conversion is performed on the date and time. The database fields are simply numeric data types that represent a date and time rather than true date and time fields. Java does not provide a method to retrieve dates or times in this format. However, it is easy to create a way to provide a method. The getRawDate() and getRawTime() methods to do this are discussed in detail later.

Any exceptions that the methods generate are simply passed back to the caller.

```

private void addOrderHeader (String CustomerNbr,
                           BigDecimal anOrderNbr,
                           BigDecimal anOrderLineCount)
throws Exception
{
    // Get the current date and time
    java.util.Date currentDateTime = new java.util.Date();

    // Open the file
    if (!ordersFile.isOpen())
    {
        ordersFile.open(AS400File.READ_WRITE,0,
                       AS400File.COMMIT_LOCK_LEVEL_DEFAULT);
    }

    // Create an empty record
    Record ordersRcd = ordersFormat.getNewRecord();

    // Set the record field values
    ordersRcd.setField("OWID", WAREHOUSE);
    ordersRcd.setField("ODID", new BigDecimal(DISTRICT));
    ordersRcd.setField("OCID", aCustomerNbr);
    ordersRcd.setField("OID", anOrderNbr);
    ordersRcd.setField("OLINES", anOrderLineCount);
    ordersRcd.setField("OCARID", "ZZ");
    ordersRcd.setField("OLOCAL", new BigDecimal(1));
    ordersRcd.setField("OENTDT",
                      new BigDecimal(getRawDate(currentDateTime)));
    ordersRcd.setField("OENTTM",

```

```

        new BigDecimal(getRawTime(currentDateTime)));
    // Add a new order header record
    ordersFile.write(ordersRcd);
    return;
}

```

### The addOrderLine() method

The complete addOrderLine() method includes the following series of events:

1. Determine the amount of discount for this customer.
2. Open the ORDLIN file and create an empty order line record.
3. Get an Enumeration object from the vector of order lines.
4. A loop is entered that continues until no more order lines can be extracted from the order. Each order line is used to populate the fields in the order line record.
5. Each record is written to the ORDLIN file.
6. The value of the order is accumulated and the stock quantity for each item is updated.
7. When all order lines have been processed, the total order value is returned to the caller.

Any exceptions that the methods generate are simply passed back to the caller.

```

private BigDecimal addOrderLine (BigDecimal anOrderNbr,
                               Order anOrder ) throws Exception
{
    BigDecimal orderTotal = new BigDecimal(0);
    int lineCounter = 0;

    // Get the customer discount percentage
    BigDecimal customerDiscount = getCustomerDiscount(anOrder.getCustomerId());

    // Open the file
    if (!orderLineFile.isOpen())
    {
        orderLineFile.open(AS400File.READ_WRITE, 0,
                          AS400File.COMMIT_LOCK_LEVEL_DEFAULT);
    }

    // Create an empty record
    Record orderLineRcd = orderLineFormat.getNewRecord();

    // Get Order detail elements from the vector
    java.util.Enumeration e = anOrder.getOrderDetail().elements();
    // While we have order lines to process
    while(e.hasMoreElements())
    {
        OrderDetail orderLine = (OrderDetail) e.nextElement();

        // Set the record field values
        orderLineRcd.setField("OLWID", WAREHOUSE);
        orderLineRcd.setField("OLDID", new BigDecimal(DISTRICT));
        orderLineRcd.setField("OLOID", anOrderNbr);
        orderLineRcd.setField("OLNBR", new BigDecimal(++lineCounter));
        orderLineRcd.setField("OLSPWH", "JAVA");
        orderLineRcd.setField("OLIID", orderLine.getItemId());
        orderLineRcd.setField("OLQTY", orderLine.getItemQty());
        BigDecimal orderAmount = (orderLine.getItemPrice()).subtract(

```

```

        orderLine.getItemPrice().multiply(customerDiscount).
            divide(new BigDecimal(100), BigDecimal.ROUND_DOWN)).
            multiply(orderLine.getItemQty()).
            setScale(2, BigDecimal.ROUND_HALF_UP);
    orderLineRcd.setField("OLAMNT", orderAmount);
    orderLineRcd.setField("OLDLVD", new BigDecimal(12311999));
    orderLineRcd.setField("OLDLVT", new BigDecimal(235959));

    // Add a new order detail record
    orderLineFile.write(orderLineRcd);

    // Accumulate the order total
    orderTotal = orderTotal.add(orderAmount);

    // Update the stock record
    updateStock(orderLine.getItemId(), orderLine.getItemQty());
}

return orderTotal;
}

```

### The **getCustomerDiscount()** method

This is the complete `getCustomerDiscount()` method. It simply opens the CSTMR file and performs a keyed read (or CHAIN in RPG) of the file. A key list is built from the customer identifier, district identifier, and warehouse identifier. If the keyed read was successful, a record is returned, and we extract the customer discount percentage from the record. This value is returned to the caller.

Any exceptions that the methods generate are simply passed back to the caller.

```

private BigDecimal getCustomerDiscount (String aCustomerID)
throws Exception
{
    // Open the file
    if (!customerFile.isOpen())
    {
        customerFile.open(AS400File.READ_WRITE, 0,
                          AS400File.COMMIT_LOCK_LEVEL_DEFAULT);
    }

    // Create a key
    Object[] customerKey = new Object[3];
    customerKey[0] = aCustomerID;
    customerKey[1] = new BigDecimal(DISTRICT);
    customerKey[2] = WAREHOUSE;

    // Perform a keyed read for the district record
    Record customerRcd = customerFile.read(customerKey);

    // If the keyed read was successful
    if (customerRcd != null)
    {
        // Extract the customer discount from the record
        BigDecimal customerDiscount = (BigDecimal)customerRcd.getField("CDCT");

        return customerDiscount;
    }
    else

```

```

    {
        return null;
    }
}

```

### The getOrderNumber() method

This is the complete `getOrderNumber()` method. This method opens the DSTRCT file and performs a keyed read using the district identifier and warehouse identifier. If a record is successfully retrieved, the order number increases and is updated in the file. The order number is returned to the caller.

**Note:** The `orderNumber.add(new BigDecimal(1))` method does not affect the value of the `orderNumber` variable. This is because the `orderNumber.add(new BigDecimal(1))` method returns a new instance of `BigDecimal` that contains the increased value. This is, of course, a temporary value that is passed directly to the `districtRecord.setField()` method.

Any exceptions that the methods generate are simply passed back to the caller.

```

private BigDecimal getOrderNumber () throws Exception
{
    System.out.println("getOrderNumber:");

    // Open the file
    if (!districtFile.isOpen())
    {
        districtFile.open(AS400File.READ_WRITE, 0,
                          AS400File.COMMIT_LOCK_LEVEL_DEFAULT);
    }

    // Create a key
    Object[] districtKey = new Object[2];
    districtKey[0] = new BigDecimal(DISTRICT);
    districtKey[1] = WAREHOUSE;

    // Perform a keyed read for the district record
    Record districtRcd = districtFile.read(districtKey);

    // If the keyed read was successful
    if (districtRcd != null)
    {
        // Extract the order number from the result set
        BigDecimal orderNumber = (BigDecimal)districtRcd.getField("DNXTOR");

        // Update the order number (positioned update)
        districtRcd .setField("DNXTOR", orderNumber.add(new BigDecimal(1)));
        districtFile.update(districtRcd);
        return orderNumber;
    }
    else
    {
        return new BigDecimal(0);
    }
}

```

## The updateCustomer() method

This is the complete updateCustomer() method. It determines the current date by creating a Date object. You must qualify the constructor, because there are two Date classes available: one in *java.util* and another in *java.sql*. Then, it opens the CSTMR file (the file may already be open from the getCustomerDiscount() method, which is why we test as if it is already open). Next, we perform a keyed read for the customer and if a record is found. We extract the current values for the customer balance and year-to-date sales. We update these fields and set the date and time of the order, and the customer record is updated in the database.

Any exceptions that the methods generate are simply passed back to the caller.

```
private void updateCustomer (String aCustomerID,BigDecimal  
                           anOrderTotal )  
throws Exception  
{  
    // Get the current date and time  
    java.util.Date currentTime = new java.util.Date();  
  
    // Open the file  
    if (!customerFile.isOpen())  
    {  
        customerFile.open(AS400File.READ_WRITE, 0,  
                          AS400File.COMMIT_LOCK_LEVEL_DEFAULT);  
    }  
  
    // Create a key  
    Object[] customerKey = new Object[3];  
    customerKey[0] = aCustomerID;  
    customerKey[1] = new BigDecimal(DISTRICT);  
    customerKey[2] = WAREHOUSE;  
  
    // Perform a keyed read for the customer record  
    Record customerRcd = customerFile.read(customerKey);  
  
    // If the keyed read was successful  
    if (customerRcd != null)  
    {  
        // Extract the current balance and year to date sales from record  
        BigDecimal currentBalance = (BigDecimal)customerRcd.  
            getField("CBAL");  
        BigDecimal yearToDateSales = (BigDecimal)customerRcd.  
            getField("CYTD");  
  
        // Update the current balance, year to date, date and time of order  
        customerRcd.setField("CBAL",currentBalance.add(anOrderTotal));  
        customerRcd.setField("CYTD",yearToDateSales.add(anOrderTotal));  
        customerRcd.setField("CLDATE",  
            new BigDecimal(getRawDate(currentDateTime)));  
        customerRcd.setField("CLTIME",  
            new BigDecimal(getRawTime(currentDateTime)));  
        customerFile.update(customerRcd);  
    }  
    return;  
}
```

## The updateStock() method

This is the complete updateStock() method. This method opens the STOCK file and performs a keyed read using the warehouse identifier and part identifier. If a record is successfully retrieved, the stock quantity decreases and is updated in the file.

Any exceptions that the methods generate are simply passed back to the caller.

```
private void updateStock (String aPartNbr,BigDecimal aPartQty )
    throws Exception
{
    // Open the file
    if (!stockFile.isOpen())
    {
        stockFile.open(AS400File.READ_WRITE, 0,
                      AS400File.COMMIT_LOCK_LEVEL_DEFAULT);
    }
    // Create a key
    Object[] stockKey = new Object[2];
    stockKey[0] = WAREHOUSE;
    stockKey[1] = aPartNbr;

    // Perform a keyed read for the district record
    Record stockRcd = stockFile.read(stockKey);

    // If the keyed read was successful
    if (stockRcd != null)
    {
        // Extract the stock quantity from the record
        BigDecimal stockQty = (BigDecimal)stockRcd.getField("STQTY");

        // Update the stock quantity
        stockRcd.setField("STQTY",stockQty.subtract(aPartQty));
        stockFile.update(stockRcd);
    }
    return;
}
```

### The writeDataQueue() method

The complete writeDataQueue() method follows this sequence of events:

1. Creates a description of the data queue layout.
  - a. It creates objects that represent the different data types that are used in the layout.
  - b. It creates an object that represents the iSeries server data queue object.
  - c. It defines the layout of each record in the data queue by using the field definitions that were created earlier.
2. Populates the record with the values for the data queue.
3. Sends the record to the data queue to initiate printing the order.

Any exceptions that the methods generate are simply passed back to the caller.

```
private void writeDataQueue (String aCustomerID,BigDecimal anOrderID )
throws Exception
{
    // Create some data type objects to describe the data queue layout
    CharacterFieldDescription as4CustomerID =
        new CharacterFieldDescription(new AS400Text(4, as400), "customerID" .);
    PackedDecimalFieldDescription as4DistrictID =
        new PackedDecimalFieldDescription(new AS400PackedDecimal(3,0), "districtID");
    CharacterFieldDescription as4WarehouseID =
        new CharacterFieldDescription(new AS400Text(4, as400), "warehouseID");
    PackedDecimalFieldDescription as4OrderID =
        new PackedDecimalFieldDescription(new AS400PackedDecimal(9,0), " .orderID");
```

```

// Create a data queue object
DataQueue dqOutput = new DataQueue(as400,
    "/" + SYSTEM_LIBRARY + "/" + DATA_QUEUE_LIBRARY + "/" + DATA_QUEUE_NAME);

// Create a record format object describing the data queue layout
RecordFormat rfOutput = new RecordFormat();
rfOutput.addFieldDescription(as4CustomerID);
rfOutput.addFieldDescription(as4DistrictID);
rfOutput.addFieldDescription(as4WarehouseID);
rfOutput.addFieldDescription(as4OrderID);

// Set up the data queue entry field values
Record recordOutput = rfOutput.getNewRecord();
recordOutput.setField("customerID", aCustomerID);
recordOutput.setField("districtID", new BigDecimal(DISTRICT));
recordOutput.setField("warehouseID", WAREHOUSE);
recordOutput.setField("orderID", anOrderID);

// Send the data queue entry
dqOutput.write(recordOutput.getContents());
return;
}

```

### The `getRawDate()` method

This method accepts a Date object and returns an unedited string representation of the date. We do this by creating our own dateFormatter object as a SimpleDateFormat. The format we require is a four-digit year, a two-digit month, and a two-digit day.

```

private String getRawDate(Date aDate)
{
    DateFormat dateFormatter = new SimpleDateFormat("yyyyMMdd");
    return(dateFormatter.format(aDate));
}

```

### The `getRawTime()` method

This method accepts a Date object and returns an unedited string representation of the time. We do this by creating our own timeFormatter object as a SimpleDateFormat. The format we require is a 24-hour clock.

```

private String getRawTime(Date aDate )
{
    DateFormat timeFormatter = new SimpleDateFormat("HHmmss");
    return(timeFormatter.format(aDate));
}

```

## 9.6.2 Cleaning up

After an order is processed, we need to perform some clean up, such as closing files, dropping the connection to the iSeries server, and so on. Java provides a standard way of doing this. We create a method called `finalize()`. If a method of this name exists in a class, the Java runtime guarantees that it is called before any garbage collection is performed on the object. This provides a convenient way to ensure that the database is left in a consistent state when we finish.

A `finalize()` method must have the following signature. It must be protected, return void, not accept arguments, and throw the `Throwable` exception, because it calls the `finalize()` method of its super class.

First, we close any open files and end commitment control. Setting each of the file objects to null explicitly allows the garbage collector to reclaim the objects. Then, we close the connection to the iSeries server and execute the `finalize()` method of the super class.

```
protected void finalize() throws Throwable
{
    // Close the file objects
    if (ordersFile.isOpen())
    {
        ordersFile.close();
        ordersFile = null;
    }
    if (customerFile.isOpen())
    {
        orderLineFile.close();
        orderLineFile = null;
    }
    if (customerFile.isOpen())
    {
        customerFile.close();
        customerFile = null;
    }
    if (stockFile.isOpen())
    {
        stockFile.close();
        stockFile = null;
    }
    if (districtFile.isOpen())
    {
        districtFile.close();
        districtFile = null;
    }

    // End commitment control
    if (    ordersFile.isCommitmentControlStarted()
          orderLineFile.isCommitmentControlStarted()
          customerFile.isCommitmentControlStarted()
          stockFile.isCommitmentControlStarted()
          districtFile.isCommitmentControlStarted()
    )
    {
        customerFile.endCommitmentControl();
    }

    // Close the AS400 system connection
    if (as400.isConnected())
    {
        as400.disconnectAllServices();
        as400 = null;
    }
    super.finalize();
    return;
}
```

There is one limitation in using the DDM classes to implement the `OrderEntry` class. They are specific to the iSeries server. If the Order Entry application needs to be transportable, it is better to use portable Java classes. We can accomplish this by using JDBC as the database access mechanism, which is discussed in the following section.

## 9.7 Order Entry using JDBC

In this section, we build the Java order processing program using JDBC. We create a class named OrderEntryJDBC, which is equivalent to the OrderEntryDDM class that was discussed previously in this chapter.

OrderEntryJDBC is a platform-independent version of the OrderEntry class, as shown in Figure 9-36. We achieve platform independence by using JDBC, which shields our application from platform unique considerations.

```
serverOrderEntry Package
  OrderEntryJDBC Class
    commitOrder()
    addOrderLine()
    addOrderHeader()
    updateStock()
    updateCustomer()
    writeDataQueue()
    getOrderNumber()
    finalize()
    initialize()
```

Figure 9-36 OrderEntryJDBC class

Three options of JDBC are available:

- ▶ JDBC through the ODBC bridge
- ▶ JDBC through the IBM Toolbox for Java
- ▶ JDBC through the IBM Developer Kit for Java

The option that we use is determined by which JDBC driver we choose to load. Loading a driver requires a URL to provide the connection information. Each of the three drivers requires a different URL:

- ▶ JDBC ODBC: "jdbc:odbc:"
- ▶ IBM Toolbox for Java JDBC: "jdbc:as400://"
- ▶ IBM Developer Kit for Java JDBC: "jdbc:db2:"

You can explicitly load JDBC drivers by registering a driver with the driver manager and connecting to the URL. Or you can implicitly load them by executing the `Class.forName()` method, for example:

```
DriverManager.registerDriver(new AS400JDBCDriver());
dbConnection =
  DriverManager.getConnection("jdbc:db2://SYSTEM1/LIB1", USER, PASSWORD);
or
Class.forName ("com.ibm.db2.jdbc.app.DB2Driver");
dbConnection =
  DriverManager.getConnection("jdbc:db2://SYSTEM1/LIB1",
  USER, PASSWORD);
```

The advantage of using the `Class.forName()` method is that a `ClassNotFoundException` exception is signaled if the requested JDBC driver cannot be found. This can create code that runs on many platforms by testing for the most specific driver and loading successively generic drivers, for example:

```

String url    = "jdbc:db2:SYSTEM";

try
{
    // Load the native JDBC driver
    Class.forName ("com.ibm.db2.jdbc.app.DB2Driver");
} catch (ClassNotFoundException e) // not found so ....
try
{
    // Load the IBM Toolbox for Java JDBC driver
    Class.forName ("com.ibm.as400.access.AS400JDBCDriver");
    url    = "jdbc:as400://SYSTEM";
} catch (ClassNotFoundException e) // not found so ....
{
    // Load the JDBC-ODBC driver
    Class.forName ("java.sql.JdbcOdbcDriver");
    url    = "jdbc:odbc:SYSTEM";
}

// Attempt to connect to a driver.  Each one of the registered drivers
// will be loaded until one is found that can process this URL

Connection con = DriverManager.getConnection (url, "my-user",
                                             "my-passwd");

```

Here is the class definition for the JDBC version. We include the JDBC routines from the java.sql package. Some of the class variables have changed to support SQL constructs.

```

package serverOrderEntry;

import com.ibm.as400.access.*; // for IBM Toolbox for Java classes (DQ
                                // support)
import java.math.*; // for BigDecimal class
import java.sql.*; // for JDBC classes
import java.util.*; // for Properties class
import java.text.*; // for DateFormat class
/**
 * This class was generated by a SmartGuide.
 *
 * This class is a replacement for the ORDENTR RPG IV program.
 * The method and variable names have been improved slightly
 * since Java supports longer names than RPG.
 */
public class OrderEntryJDBC
{
    // Mnemonic values
    private static final String SYSTEM_LIBRARY = "QSYS.LIB";
    private static final String DATA_QUEUE_NAME = "ORDERS.DTAQ";
    private static final String DATA_QUEUE_LIBRARY = "your-library.LIB";
    private static final String WAREHOUSE = "0001";
    private static final int DISTRICT = 1;
    private static final String SYSTEM = "localhost";
    private static final String USER = "your-user-id";
    private static final String PASSWORD = "your-password";
    private static final String DATA_LIBRARY = "your-library";

    // an AS400 system object for DataQueue support
    private AS400 as400 = null;

    // A global connection and prepared statement
    private Connection dbConnection = null;
    private PreparedStatement psAddOrderLine = null;

```

```

// Create an executable SQL statement object
private Statement addOrderHeader = null;
private Statement getDiscount = null;
private Statement getOrderNumber = null;
private Statement setOrderNumber = null;
private Statement getCustomer = null;
private Statement setCustomer = null;
private Statement getStockQty = null;
private Statement setStockQty = null;
}

```

### 9.7.1 Method logic

The method logic for the JDBC version of the OrderEntry class is similar to the DDM version. Only the implementation differs. We use JDBC methods instead of DDM methods here.

#### The initialize() method

This is the complete JDBC `initialize()` method. This method is run by the constructor for the OrderEntry class. It uses a different technique to create a connection to the iSeries server. A JDBC properties object describes the attributes of the connection.

JDBC allows either a string of properties to be specified in the URL for the connection or a properties object to be used in addition to the URL. Using the properties object allows a little more flexibility in defining the connection, because it can be encapsulated in another class.

We create a connection to the iSeries server for use by the data queue methods in `writeDataQueue()`. It is better to connect using a global variable, because the connection can be time consuming. It is not a good idea to connect and disconnect frequently.

We create the properties object and set a number of the property values. The values for `USER` and `PASSWORD` are named constants found in the class definition.

**Note:** It is necessary to provide a user ID and password even when running on the same iSeries server as the database to which you are connected. You can use the value `*current` for the user ID and password. In this case, the user ID and password for the current iSeries session is used.

Then, we use the `Class.forName()` class method to determine the proper JDBC driver, and automatically register and load it. Next, we create a connection to the iSeries server by using our properties object.

This block of code creates a prepared statement for the only SQL statement that is run repeatedly. A prepared statement is more efficient than running a dynamic SQL statement.

```

private void initialize () throws Exception
{
    // Create an AS400 system connection object
    as400 = new AS400(SYSTEM, USER, PASSWORD);

    // Create a properties object for JDBC connection
    Properties jdbcProperties = new Properties();

    // Set the properties for the JDBC connection
    jdbcProperties.put("user", USER);
    jdbcProperties.put("password", PASSWORD);
    jdbcProperties.put("naming", "sql");
}

```

```

jdbcProperties.put("errors", "full");
jdbcProperties.put("date format", "iso");

// Load the AS400 system Native JDBC driver into the JVM
// This method automatically verifies the existence of the driver
// and loads it into the JVM—should not use
// DriverManager.registerDriver()
Class.forName ("com.ibm.db2.jdbc.app.DB2Driver");

// Connect using the properties object
dbConnection =
    DriverManager.getConnection("jdbc:db2://"+SYSTEM+"/"+DATA_LIBRARY,
                               jdbcProperties);

// Prepare the ORDLIN SQL statement
psAddOrderLine = dbConnection.prepareStatement("INSERT INTO "+
    "ORDLIN (OLOID, OL DID, OL WID, OLNBR, OLSPWH, OLIID," +
    "OLQTY, OLAMNT, OLDVD, OLDLVT) "+
    "VALUES(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");

// Create an executable statement
getOrderNumber = dbConnection.createStatement();
setOrderNumber = dbConnection.createStatement();
getStockQty = dbConnection.createStatement();
setStockQty = dbConnection.createStatement();
getCustomer = dbConnection.createStatement();
setCustomer = dbConnection.createStatement();
getCustomer.setCursorName("CUSTOMER");
getOrderNumber.setCursorName ("DISTRICT");
getStockQty.setCursorName("STOCK");
return;
}

```

## The commitOrder() method

The method of committing changed data is altered. A commit request is placed through the connection object, rather than through any of the files.

```

public String commitOrder (Order anOrder) throws Exception
{
    try
    {
        // Extract the customer number and count of lines
        String customerNumber = anOrder.getCustomerId();
        BigDecimal orderLineCount = new BigDecimal(anOrder.getOrderDetail().size());

        // Determine the order number
        BigDecimal orderNumber = getOrderNumber();

        // Add the line items to the order detail file
        BigDecimal orderTotal = addOrderLine(orderNumber, anOrder);

        // Add the order header
        addOrderHeader(customerNumber, orderNumber, orderLineCount);

        // Update the customer
        updateCustomer(customerNumber, orderTotal);

        // Commit the database changes
        if (dbConnection.getTransactionIsolation() !=
            java.sql.Connection.TRANSACTION_NONE)
        {

```

```

        dbConnection.commit();
    }

    // Initiate order printing
    writeDataQueue(customerNumber, orderNumber);
}
catch(Exception e)
{
    e.printStackTrace();
    return("Order processing failed.");
}
return("Order processed successfully.");
}

```

### The addOrderHeader() method

The only change necessary in this method is to replace the field population and write statement with an SQL INSERT statement.

**Note:** The SQL statement is built in a variable and passed to the executeUpdate() method. We do this, so we can see the final SQL statement with debug. If we pass the SQL statement as literal strings, it is difficult to find SQL syntax errors.

```

private void addOrderHeader (String aCustomerNumber,
                           BigDecimal anOrderNumber,
                           BigDecimal anOrderLineCount)
throws Exception
{
    // Get the current date and time
    java.util.Date currentDate = new java.util.Date();

    // Create an executable statement
    addOrderHeader = dbConnection.createStatement();

    // Add a new order header record
    String sql = "INSERT INTO ORDERS " +
        "(OWID, ODID, OCID, OID, OLINES, OCARID, OLOCAL, OENTDT,
        OENTTM)+" +
        "VALUES ('"+WAREHOUSE+"'" +
        ", "+DISTRICT+
        ", '"+aCustomerNumber+"'" +
        ", "+anOrderNumber.toString()+" +
        ", "+anOrderLineCount.toString()+" +
        ", 'ZZ', 1"+
        ", "+getRawDate(currentDate)+
        ", "+getRawTime(currentDate)+")";

    addOrderHeader.executeUpdate(sql);
    return;
}

```

### The addOrderLine() method

The only change necessary in this method is to replace the field population and write statement with an SQL INSERT statement. We use an SQL prepared statement in this method because there may be multiple order lines that result in the same SQL statement being run multiple times. Preparing the SQL statement before using it is faster if the statement is run many times.

**Note:** We can also see some performance improvements by changing the other methods to use prepared statements.

```
private BigDecimal addOrderLine (BigDecimal anOrderNumber,
                                Order anOrder) throws Exception
{
    BigDecimal orderTotal = new BigDecimal(0);
    int lineCounter = 0;

    // Get the customer discount percentage
    BigDecimal customerDiscount = getCustomerDiscount(anOrder.getCustomerId());

    // Get Order detail elements from the vector
    java.util.Enumeration e = anOrder.getOrderDetail().elements();
    // While we have order lines to process
    while(e.hasMoreElements())
    {
        OrderDetail orderLine = (OrderDetail) e.nextElement();

        // Set the parameter markers for the SQL statement
        psAddOrderLine.setBigDecimal(1, anOrderNumber);
        psAddOrderLine.setBigDecimal(2, new BigDecimal(DISTRICT));
        psAddOrderLine.setString(3, WAREHOUSE);
        psAddOrderLine.setBigDecimal(4, new BigDecimal(++lineCounter));
        psAddOrderLine.setString(5, "JAVA");
        psAddOrderLine.setString(6, orderLine.getItemId());
        psAddOrderLine.setBigDecimal(7, orderLine.getItemQty());
        BigDecimal orderAmount = (orderLine.getItemPrice().subtract(
            orderLine.getItemPrice().multiply(customerDiscount).
            divide(new BigDecimal(100), BigDecimal.ROUND_DOWN))).
            multiply(orderLine.getItemQty());
        setScale(2, BigDecimal.ROUND_HALF_UP);

        psAddOrderLine.setBigDecimal(8, orderAmount);
        psAddOrderLine.setBigDecimal(9, new BigDecimal(12311999));
        psAddOrderLine.setBigDecimal(10, new BigDecimal(235959));

        // Add the order line record
        psAddOrderLine.executeUpdate();

        // Accumulate the order total
        orderTotal = orderTotal.add(orderAmount);

        // Update the stock record
        updateStock(orderLine.getItemId(), orderLine.getItemQty());
    }
    return orderTotal;
}
```

### The **getCustomerDiscount()** method

This method is similar to the DDM version. We create an SQL statement object and build an SQL query that returns the values in which we are interested. In this case, this is only the customer discount field (CDCT).

We run the query, which returns a result set. A result set contains the rows that are retrieved by the query. There may be many rows returned, in which case a loop processes the result set. Here we expect only one row and do no looping. The `ResultSet` class provides a `next()` method to fetch the rows from the result set. If no rows exist or no more rows are available (equivalent to end-of-file), null is returned.

After we have retrieved the row from the result set, we extract the value for the customer discount field. The `ResultSet` class provides methods for extracting column values that are appropriate to each database data type. Here we use the `getBigDecimal()` method because the CDCT is a packed decimal field.

We return the value of the customer discount to the caller.

```
private BigDecimal getCustomerDiscount (String aCustomerID) throws Exception
{
    // Create an executable statement
    getDiscount = dbConnection.createStatement();

    // Get the customer record
    String sql = "SELECT CDCT FROM CSTMR WHERE CID = '" +aCustomerID+ "'";

    ResultSet rs = getDiscount.executeQuery(sql);

    // Extract the customer discount from the result set
    rs.next();
    BigDecimal customerDiscount = rs.getBigDecimal("CDCT");

    return customerDiscount;
}
```

If we need to process many rows from the result set, the Java code looks similar to this example:

```
while( rs.next() )
{
    // do row processing
}
```

### The `getOrderNumber()` method

This method has the same structure as the DDM version. However, we need to do some additional processing to update a row. You can update records by specifying selection criteria on the WHERE clause or by doing a positioned update where the record just read is the one updated. This is more efficient than using the WHERE clause in this case. A positioned update can only be done through a named SQL cursor. A statement can only be named once, so we either must catch the exception as shown here, or define and name the SQL statements globally. Both techniques have drawbacks, which include a performance penalty for raising the exception or a maintenance concern with global variables.

Two SQL statement objects were already created and named when initializing:

- ▶ `getOrderNumber` for fetching the records
- ▶ `setOrderNumber` for updating the records

We named the cursor, so we can reference it in the UPDATE statement.

We build an SQL query to retrieve the row that contains the next order number. We run the query, fetch the row from the result set, and extract the field value. Next, we build an SQL UPDATE statement to increment the order number value and update the database.

Finally, we return the order number to the caller.

```
private BigDecimal getOrderNumber () throws Exception
{
    System.out.println("getOrderNumber:");

    // Get the next available order number
    String sql = "SELECT DNXTOR FROM DSTRCT "+
        "WHERE DID = "+DISTRICT+" AND DWID = '"+WAREHOUSE+" "+
        "FOR UPDATE";

    ResultSet rs = getOrderNumber.executeQuery(sql);

    // Extract the order number from the result set
    rs.next();
    BigDecimal orderNumber = rs.getBigDecimal("DNXTOR");

    // Update the order number (positioned update)
    sql = "UPDATE DSTRCT "+
        "SET DNXTOR='"+ orderNumber.add(new BigDecimal(1)).toString() +
        " WHERE CURRENT OF DISTRICT";

    setOrderNumber.executeUpdate(sql);

    return orderNumber;
}
```

### The updateCustomer() method

This is another method that uses the positioned update technique that was described earlier. Again, two SQL statement objects were already created and named when initializing:

- ▶ getCustomer for fetching the records
- ▶ setCustomer for updating the records

We build and run a query, extract values from the result set, and build and run an UPDATE statement.

The customer file is updated with the date and time of the order, current balance, and total year-to-date sales.

```
private void updateCustomer (String aCustomerID,
                           BigDecimal anOrderTotal ) throws Exception
{
    // Get the current date and time
    java.util.Date currentDate = new java.util.Date();

    // Get the customer record
    String sql = "SELECT * FROM CSTMR "+
        "WHERE CID = '"+aCustomerID+"' "+
        "FOR UPDATE OF CLDATE, CLTIME, CBAL, CYTD";

    ResultSet rs = getCustomer.executeQuery(sql);

    // Extract the current balance and year to date sales from the result set
    rs.next();
    BigDecimal currentBalance = rs.getBigDecimal("CBAL");
    BigDecimal yearToDateSales = rs.getBigDecimal("CYTD");

    // Update the current balance and year to date sales (positioned update)
    sql = "UPDATE CSTMR "+
        "SET CLDATE = "+getRawDate(currentDate)+
```

```

    ", CLTIME = "+getRawTime(currentDateTime)+
    ", CBAL = "+(currentBalance.add(anOrderTotal)).toString()+
    ", CYTD = "+(yearToDateSales.add(anOrderTotal)).toString()+
    " WHERE CURRENT OF CUSTOMER";

setCustomer.executeUpdate(sql);

return;
}

```

### **The updateStock() method**

Again, we use the positioned update technique that was described earlier. Two SQL statement objects were already created and named when initializing:

- ▶ getStockQty for fetching the records
- ▶ setStockQty for updating the records

We build and run a query, extract values from the result set, and build and run an UPDATE statement.

In this case, we subtract the number of items sold from the current quantity and update the database.

```

private void updateStock (String aPartNbr,
                        BigDecimal aPartQty ) throws Exception
{
    // Get the next available order number
    String sql = "SELECT STQTY FROM STOCK " +
                 "WHERE STIID = '" +aPartNbr.toString()+"' AND STWID  = '" +WAREHOUSE+"'" +
                 "FOR UPDATE";

    ResultSet rs = getStockQty.executeQuery(sql);

    // Extract the order number from the result set
    rs.next();
    BigDecimal stockQty = rs.getBigDecimal("STQTY");

    // Update the order number (positioned update)
    sql = "UPDATE STOCK "+
          "SET STQTY = " + stockQty.subtract(aPartQty).toString() +
          " WHERE CURRENT OF STOCK";

    setStockQty.executeUpdate(sql);

    return;
}

```

### **The writeDataQueue() method**

No changes are required to this method.

### **The getRawDate() method**

No changes are required to this method.

### **The getRawTime() method**

No changes are required to this method.

**Note:** Because these three methods are common to both OrderEntry classes, it makes more sense to encapsulate them in an object. You can perform this task as an exercise.

### 9.7.2 Cleaning up

This method closes cursor objects rather than file objects. Otherwise, it is similar to the DDM version.

```
protected void finalize() throws Throwable
{
    // Close the cursor objects
    addOrderHeader.close();
    addOrderHeader = null;
    psAddOrderLine.close();
    psAddOrderLine = null;
    getCustomer.close();
    getCustomer = null;
    setCustomer.close();
    setCustomer = null;
    getStockQty.close();
    getStockQty = null;
    setStockQty.close();
    setStockQty = null;
    getOrderNumber.close();
    getOrderNumber = null;
    setOrderNumber.close();
    setOrderNumber = null;

    // Close the AS400 system connection
    if (!dbConnection.isClosed())
    {
        dbConnection.close();
        dbConnection = null;
    }

    // Close the AS400 system object
    if (as400.isConnected())
    {
        as400.disconnectAllServices();
        as400 = null;
    }

    super.finalize();
    return;
}
```

## 9.8 Remote method invocation support

At this point, we successfully created two Java classes that are functionally equivalent to the RPG Order Entry program with which we started. If we create an Order object and pass it to either of the OrderEntry classes, we can test the classes. However, these classes are meant to be run from a client front end. We cannot do that yet, because there is no linkage to the client.

Remote method invocation is the mechanism that Java uses to support running methods on physically separate systems. A TCP/IP connection must exist between the systems, and certain applications must be running to support RMI, specifically the RMI registry.

To make RMI work, the following points must be true:

1. The class must be a subclass of the UnicastRemoteObject class.
2. The class must implement an interface that describes the public methods.
3. The interface must be a subclass of the Remote class.
4. The interface must describe each public method.
5. The interface methods must throw RemoteException.
6. An RMI registry must be running on the server.
7. An instance of the class must register with the registry.
8. The client and server must know on which TCP port to find the registry.

Here is the definition of the interface class. It does not need to be called OrderEntryI. However, such a naming convention helps keep the links between the classes clear. The class must import the java.rmi package to use the RMI classes. The class must satisfy rules 3, 4, and 5 above.

```
package orderObjects;

import java.rmi.*;
/**
 * This interface was generated by a SmartGuide.
 *
 */
public interface OrderEntryI extends Remote
{
    public String commitOrder(Order anOrder) throws RemoteException;
}
```

### 9.8.1 RMI application design

We use the Java RMI interface to allow the client Java program to interface with the server Java code. The Java client program interfaces to the iSeries program (OrderEntryDDM or OrderEntryJDBC) through a class named OrderSubmitter, as shown in Figure 9-37. An Order object is passed from the client program to the server program.

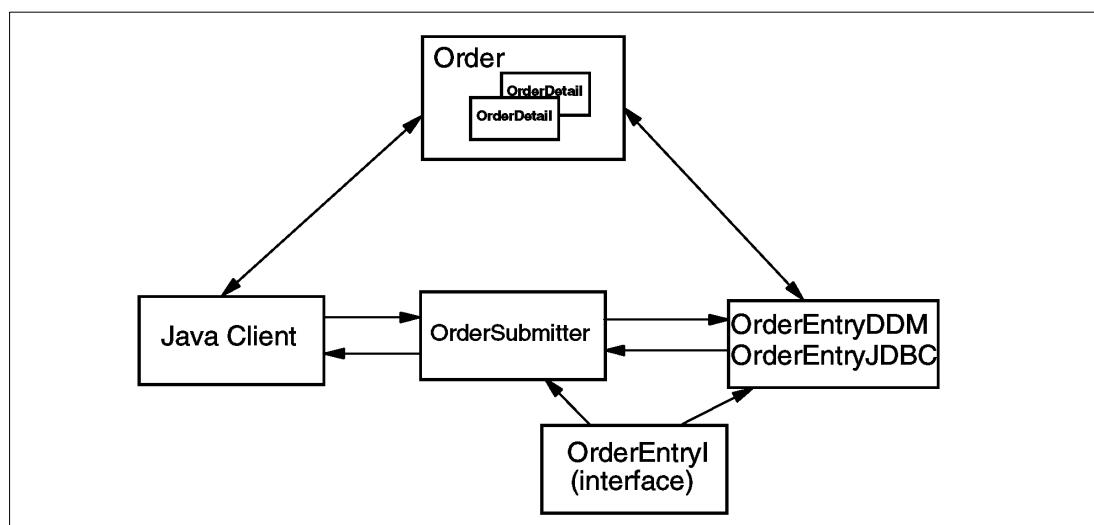


Figure 9-37 RMI application design

The public method that we use is called `commitOrder`. It is described in the interface that we implement, which is named `OrderEntryI`. The host Java program implements a method named `commitOrder`. The client programs calls this method through the `OrderSubmitter` class, as shown in Figure 9-38.

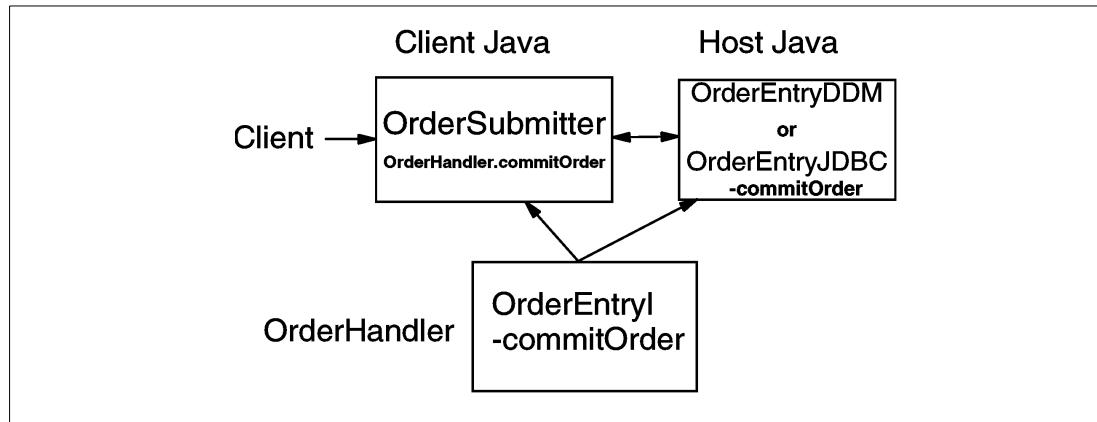


Figure 9-38 RMI Interface

### 9.8.2 Adding RMI support to a server class

Each of the `OrderEntry` classes must import a number of RMI packages to successfully support RMI. Each class must satisfy rules 1 and 2 as noted on page 398.

```

import java.rmi.*; // for Remote Method Invocation
import java.rmi.server.*;

public class OrderEntryJDBC extends UnicastRemoteObject implements
    OrderEntryI

```

Because the interface method for `commitOrder()` states that `RemoteException` is thrown, the actual `commitOrder()` implementation must also throw `RemoteException`:

```

public String commitOrder (Order anOrder) throws RemoteException
{
    try
    {
        // code removed for clarity
    }
    catch(Exception e)
    {
        e.printStackTrace();
        return("Order processing failed.");
    }
    return("Order processed successfully.");
}

```

Each remote class must be capable of registering its services with an RMI registry that provides brokering services between the client and the server. We do this by adding a main method that performs the registration. The RMI classes throw exceptions, so we must wrap our use of these classes in a `try{} catch{}` block. Here is the `main()` method from the `OrderEntryJDBC` class:

```

public static void main(String[] parameters)
{
    // Set up the server
    try

```

```

    {
        OrderEntryJDBC oeJDBC = new OrderEntryJDBC();
        Naming.rebind("//"+SYSTEM+port+"/OrderEntryServer", oeJDBC);
    } catch(Exception e) {e.printStackTrace();}
    return;
}

```

This block of code performs these functions:

- ▶ Creates a new instance of our own class
- ▶ Binds the new object as a service with an appropriate name (in this case, OrderEntryServer)

The next step, in supporting RMI, is to create stub and skeleton classes that provide the communications support. You can automatically create these by using the `rmic` command. You can also use VisualAge for Java to create these classes.

**Note:** In the Java 2 implementation of RMI, the skeleton is obsolete. VisualAge for Java will not generate this class.

Once you create the proxies, you must run the RMI registry. This must be done from the shell. Use the `rmiregistry` shell command to execute the registry.

You can direct the RMI registry to a specific TCP port, but it is probably best to let it use the default port of 1099. If you want to assign a name to the port, you can do so by adding a service table entry.

```
ADDSRVTBLE SERVICE('rmiregistry') PORT(1099) PROTOCOL('tcp')
    TEXT('Java RMI registry Service')
```

If you choose to execute the registry on a different port number, you must ensure that both the server and client use the same port for registry services. You can accomplish this by adding the port number to the `Naming.bind()` or `Naming.rebind()` methods, for example:

```
Naming.rebind("//"+SYSTEM+":55555/OrderEntryServer", oeJDBC);
```

This code specifies that the registry is using port 55555.

The final step before the remote class can be used is to actually start it. This can also be done from the shell, although it must be a different session from the one running the registry itself.

```
java serverOrderEntry.OrderEntryJDBC
```

### 9.8.3 Adding RMI support to the client

In Chapter 8, “Migrating the user interface to the Java client” on page 321, we analyze how the Java client submits an order. It creates buffers of string data that are used as parameters in a Distributed Program Call. The `ProgramCall` class, as well as other classes in the IBM Toolbox for Java, were used to do this. Admittedly, the process is somewhat complicated. This is rooted in the fact that an object-oriented program is calling a legacy program. Since a legacy program does not understand the concept of an object, objects cannot be passed as parameters to legacy routines. The objects must be “flattened” or streamed out as byte data.

However, as we saw in Chapter 8, this introduces more complexity to the program. The `Order` and `OrderDetail` classes provide `toString()` methods that helped deal with this situation. Once the server code is implemented in Java, the client task of passing parameters is simplified. The client can simply pass the `Order` object as a parameter. Rather than call a

distributed program, the client can now make use of Java RMI architecture. In previous sections, we saw how to convert some of the legacy RPG code to Java classes. These classes are designed, so they implement the RMI interfaces. The client can now run the remote methods that are made available through these new classes.

#### 9.8.4 Creating a client class to handle RMI

At the client, we create a new class to handle the RMI interface to the server. The name of this class is OrderSubmitter:

```
import java.rmi.*;  
  
public class OrderSubmitter  
{  
    private String serverURL = null;  
    private OrderEntryI orderHandler = null;  
}
```

This class contains only two instance variables. The serverURL is a string that represents a concatenation of the host machine and the name of the RMI service that has been registered on the host machine. The orderHandler is declared to be of the interface type that the host RMI class implements. See 9.8.1, “RMI application design” on page 398, for details on the OrderEntryI and OrderEntryJDBC classes. Here is code of the constructor.

```
public OrderSubmitter (String hostServer, String port)  
{  
    // define the RMI server that we are connecting to  
    serverURL = "//"+hostServer+":"+port+"/OrderEntryServer";  
}
```

The hostServer and port are passed in as parameters. These are the same values that were retrieved from the sign-on dialog discussed in Chapter 8, “Migrating the user interface to the Java client” on page 321. The ‘OrderEntryServer’ string is the registered name as the service name on host machine. See 9.8.2, “Adding RMI support to a server class” on page 399, for details on the OrderEntryJDBC class.

The linked() method handles a reference to the remote object by executing the Naming.lookup() method. Once completed, the application can run a remote method.

```
public boolean linked()  
{  
    // obtain reference to the remote object  
    try  
    {  
        orderHandler = (OrderEntryI)Naming.lookup(serverURL);  
    }  
    catch(Exception e)  
    {  
        e.printStackTrace();  
        return(false);  
    }  
    return(true);  
}
```

The OrderSubmitter class embeds the RMI in its own submit() method:

```
public String submit(Order theOrder)  
{  
    try  
    {  
        return orderHandler.commitOrder(theOrder);  
    }
```

```

        }
    catch(Exception e)
    {
        e.printStackTrace();
        return "Error invoking remote method";
    }
}

```

The submit() method simply executes the commitOrder() method on the remote object. The commitOrder() method returns a status message that the client displays in the Order Entry Window. Using RMI to submit the order simplifies the parameter passing. The Order object is now passed without having to stream the data members.

To implement the OrderSubmitter object, we have to add a new remoteSubmit() method into the Java client OrderEntryWdw.

```

private void remoteSubmit(Order theOrder)
{
    OrderSubmitter remoteRequestor =
        new OrderSubmitter(systemName, getPortTF().getText());
    if(remoteRequestor.linked())
    {
        updateStatus("Remote method server linked...");
    }
    else
    {
        updateStatus("Could not link to remote method server");
        return;
    }

    updateStatus(remoteRequestor.submit(theOrder));

    return;
}

```

We also change the retrieveOrderInfo() method to support RMI. The remoteSubmit() method will be invoked if the port TextField is entered in the sign-on dialog.

```

private void retrieveOrderInfo()
{
    int numEntries = getDefaultTableModel1().getRowCount();

    Order theOrder = new Order(getCustIdTF().getText());

    for(int i=0;i<numEntries;i++)
    {
        OrderDetail detail = new OrderDetail(
            (String)getDefaultTableModel1().getValueAt(i,0),
            (String)getDefaultTableModel1().getValueAt(i,1),
            new BigDecimal((String)getDefaultTableModel1().getValueAt(i,2)),
            new BigDecimal((String)getDefaultTableModel1().getValueAt(i,3))
        );
        theOrder.addEntry(detail);
    }

    // When port is enterd, RMI will be invoked
    // otherwise DPC instead.
    if (getPortTF().getText().equals("")) submitOrder(theOrder); // DPC
    else remoteSubmit(theOrder); // RMI
}

```

```
    return;  
}
```

Since the Order object is passed as a parameter in a RMI, it must be changed so that it implements Serializable. The Java Serializable interface, however, requires no methods that must be implemented by the user. You simply have to declare that the class implements Serializable.

Any object that is contained in the Order class must also implement the Serializable interface. The Order class contains an array of OrderDetail objects. Therefore, OrderDetail must also implement Serializable. The rest of the data members in OrderDetail and Order are not user-defined classes, so no further changes have to be made. The new declaration of the Order class is shown:

```
public class Order implements java.io.Serializable  
{  
    private String customerId;  
    private java.util.Vector orderDetail = new java.util.Vector();  
}
```

The declaration of the OrderDetail class changes in a similar fashion. Simply add '**implements Serializable**' to the class declaration. This completes the changes that are necessary to enable the client use of RMI.

## 9.9 Conclusion

To allow the Java client to interface with the new iSeries server Java classes, we use RMI. This section examined the changes that are necessary to the client code and the server code to support RMI. It covered implementing RMI for the JDBC example (OrderEntryJDBC). You can also use the same methodology to implement an RMI interface between the Java client and the OrderEntryDDM class. The advantages of using RMI are:

- ▶ Both the client and server application are written in Java. The programmer only needs to work in one language.
- ▶ The Java code is platform independent. It is easy to write Java code that can run on either the client or server.
- ▶ Calling the remote method is transparent. Calling a method using RMI is the same as calling a local method. This makes it easy to extend or modify programs by moving the actual location of the methods being used.
- ▶ Objects can be passed between the client and server code. Even though a method may reside on a remote platform, objects can be passed to it and received back from it. This makes it much easier to interface with programs running on other platforms than passing parameters. When passing parameters, we have to implement platform-unique solutions to pass the parameters from the client to the host.





## Structured Query Language for Java (SQLJ)

SQLJ is the Structured Query Language embedded in Java. It was accepted in 1998, by the American National Standards Institute (ANSI) as a standard (ANSI X3.135.10-1998). Major database vendors, which include Oracle, IBM, Sybase, Informix, JavaSoft, and other leading industry vendors, participated to develop this common standard. It provides application developers with an option to develop open, multi-vendor enterprise database applications using SQL and Java.

## 10.1 Using SQLJ

This chapter explains using SQLJ on the iSeries server through these examples:

► **Java class: CstmrInq**

A simple Java program that uses SQLJ to retrieve one record from the customer master file. Since we use Java and SQLJ, the application is platform independent. We demonstrate this by running it both on an iSeries server and a personal computer.

► **Java class: CstmrList**

A simple Java program that uses SQLJ to retrieve a group of records from the customer master file. Again, we demonstrate platform independence by running the application on an iSeries server and a personal computer.

► **Order Entry client application**

We convert the Order Entry client application, discussed in Chapter 8, “Migrating the user interface to the Java client” on page 321, to use SQLJ. The original application uses a combination of JDBC stored procedures and distributed data management (DDM) record-level access. We convert it to use SQLJ for all database access. We convert three Java classes to use SQLJ:

- OrderEntryWdw
- SltCustWdw
- SltItemWdw

► **Order Entry server application**

We convert the host JDBC Java application discussed in Chapter 9, “Moving the server application to Java” on page 349, to use SQLJ. The Java program that we create is named OrderEntrySQLJ.

This chapter provides a brief introduction to SQLJ, with an emphasis on SQLJ functionality.

## 10.2 Introduction to SQLJ

SQLJ is based on the JDBC standard. The JDBC driver establishes the initial connection between the Java program and the database server. You can use SQLJ for any database that has a JDBC driver.

The SQLJ standard has three components:

- The SQLJ file that contains embedded SQL statements in the Java program
- The translator translates the SQLJ file to produce .java files and profiles that use the runtime environment to perform SQL operations
- The runtime environment that usually performs the SQL operations in JDBC and uses the profile to obtain details about database connections

The SQLJ documentation states that SQLJ is a static implementation of SQL. Static SQL means that all of the variables that the SQL statement uses are known at compile time. On the iSeries server, static SQL also means that an access plan is created at compile time and included with the program. The access plan contains information about how to run the SQL statement. This is not the case with SQLJ on the iSeries server. Compiling an SQLJ program does not generate an access plan.

Consider these points when you compare SQLJ and JDBC:

- ▶ SQLJ source programs are smaller than the equivalent JDBC programs.
- ▶ SQLJ programs allow direct embedding of Java bind expressions within SQL statements. JDBC requires a separate call statement for each bind variable and specifies the binding by position number.
- ▶ SQLJ provides strong type checking of query output and returns parameters on pre-compile. JDBC passes values to and from SQL without compile-time type checking.
- ▶ SQLJ provides simplified rules for SQL statements and for calling stored procedures and functions. JDBC requires more statements and takes more time and effort to code.
- ▶ SQLJ generates JDBC statements. You do not have complete control of the JDBC statements as you do when writing it directly.
- ▶ If performance is your major concern, consider using JDBC rather than SQLJ because you can deal directly with the JDBC statements.

### 10.2.1 An overview of how SQLJ works

This is an overview of how to program using SQLJ and Java. The SQL statement is written in the Java source within curly brackets ({} ) and it is preceded by a symbol #sql , for example:

```
#sql { select CFIRST into :customerName from CSTMR where CID = :cstno }
```

This is all you need to enter to retrieve a customer record from the CSTMR file. This is much simpler than JDBC and is a more natural way of coding SQL statements.

To use SQLJ, the member type of the Java source is different and needs to be converted into Java code using a translator. This translator converts the SQL statement into JDBC statements. For example, we have the following .sqlj source file:

```
HelloWorld.sqlj
```

This member type needs to be translated by the SQLJ translator, for example:

```
Sqlj HelloWorld.sqlj
```

As a result of running the translator, the following files are generated:

- ▶ HelloWorld.java
- ▶ HelloWorld.class
- ▶ HelloWorld\_SJProfileKeys.class
- ▶ HelloWorld\_SJProfile0.ser

The Java source and class files are standard Java files. The translator generates the additional class and .ser file to handle the conversion of the SQL statements to JDBC statements.

VisualAge for Java also provides an SQLJ tool that implements the SQLJ standard. The translator component is integrated into the IDE, so you can import, translate, and edit SQLJ files. The runtime environment is an installable feature that is added to your workspace. The original .sqlj source files are maintained in your project resources directory, as are the profiles.

In our examples, we use VisualAge for Java as a tool to store source files, translate source files, and act as a runtime environment.

## 10.2.2 SQLJ and the iSeries server

The SQLJ translator, which was developed by Oracle Corporation, is known as the Oracle Reference Interpreter (RI). It is provided by IBM in OS/400 V4R4 and later as part of the operating system. The RI is composed of two packages and is located in the following integrated file system directories:

- ▶ /QIBM/ProdData/Java400/ext/runtime.zip
- ▶ /QIBM/ProdData/Java400/ext/translator.zip

In V5R1, the packages are moved to:

- ▶ /QIBM/ProdData/OS400/Java400/ext/runtime.zip
- ▶ /QIBM/ProdData/OS400/Java400/ext/translator.zip

The translator.zip package contains the class that contains the actual translator. This class is named `Sqlj`.

The other package, runtime.zip, contains runtime support for SQLJ classes and must be imported into each class in which you use SQLJ. SQLJ programs provide complete platform independence. A program that is compiled on the iSeries server also works on a PC without any changes.

## 10.2.3 Additional information

As mentioned in the introduction, SQLJ was developed by efforts of many software vendors. However, the translator, which is also available in the public domain free of charge, was developed by Oracle Corporation.

To find detailed information and documentation about SQLJ, go to:  
<http://www.oracle.com/java/sqlj/index.html>

## 10.3 The Cstmrlnq example

`Cstmrlnq` is a Java program that reads the customer table (CSTMRL) on the iSeries server using SQL. The program performs the following actions:

- ▶ Establishes a connection to the iSeries server host database server using a JDBC driver. If the program is running on an iSeries server, it loads the iSeries native JDBC driver. If it is running on a PC, it loads the IBM Toolbox for Java JDBC driver.
- ▶ Sets the `DefaultContext` class, which is a requirement to use SQLJ.
- ▶ Reads the CSTMRL file using SQL and stores the result in host variables.
- ▶ Writes the information read from the CSTMRL file to the display.

### 10.3.1 Setting up VisualAge for Java for the Cstmrlnq example

Before we can use SQLJ in our project, we need to add the SQL Runtime Library feature to our workspace. We can import and translate SQLJ files without the SQLJ Runtime Library feature. However, we will not be able to compile the files.

To add the SQLJ Runtime Library feature, we perform these steps:

1. In the Workbench, or a browser, select **File-> Quick Start**, and then select **Features**.
2. Select **Add Feature**, and click **OK**.
3. Select **SQLJ Runtime Library**, and click **OK**.

Then we add the SQLJ Runtime Library project to our project's class path.

We cannot add embedded SQLJ statements directly to our source code in a source pane. We have to create the *Cstmrlnq.sqlj* file on our file system with a text editor. Once we create an SQLJ file, we need to import it into the project and translate the file, before we can use it. In our example, the project is named *SQLJ Example*. See Figure 10-1.

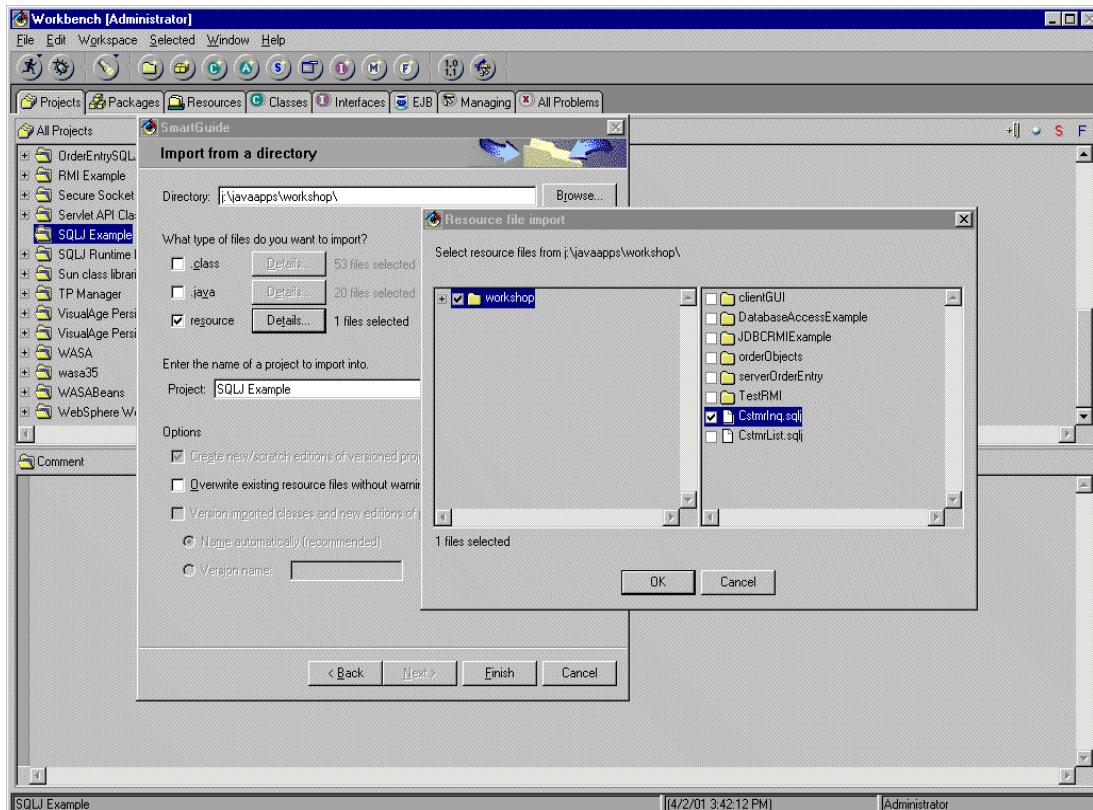


Figure 10-1 Importing the .sqlj file to the project resources directory

After the imported SQLJ file is placed in our project resources directory, we use VisualAge for Java to translate the SQLJ file using the following steps:

1. Double-click the **SQLJ Example** project, and select the **Resources** tab (Figure 10-2 on page 410).

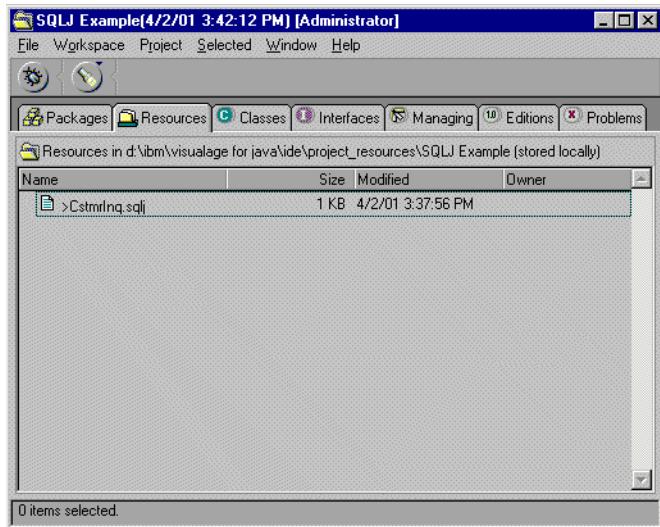


Figure 10-2 Project resources

2. Right-click the SQLJ file, and select **Tools-> SQLJ-> Translate** (Figure 10-3).

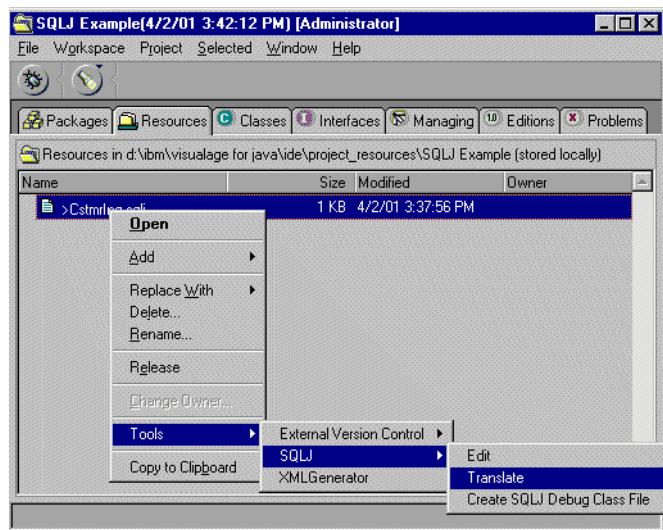


Figure 10-3 Translating the SQLJ file

3. The SQLJ translator creates two Java classes and one profile. The Java classes are in our project, and the profile is placed in the project resources directory (Figure 10-4).
  - Cstmrlnq
  - Cstmrlnq\_SJProfileKeys
  - Cstmrlnq\_SJProfile0.ser

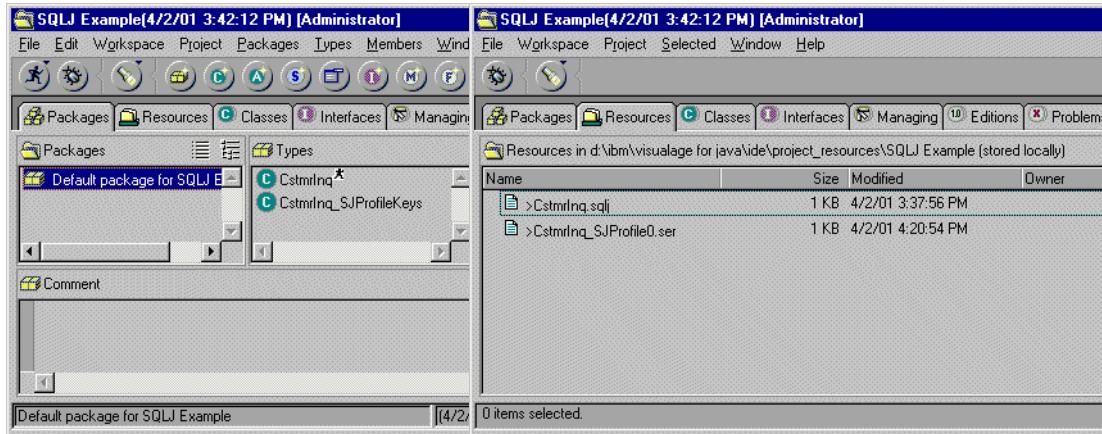


Figure 10-4 Classes and resources in the SQLJ example

### 10.3.2 Running CstmrInq

We can run the **CstmrInq** class in VisualAge for Java or export the classes and resource file to run on other systems. This section explains both scenarios.

#### Running CstmrInq in VisualAge for Java

To run the program in VisualAge for Java, select the **CstmrInq** class. Right-click and select **Run-> Run main**. Figure 10-5 shows the console after we run the program.

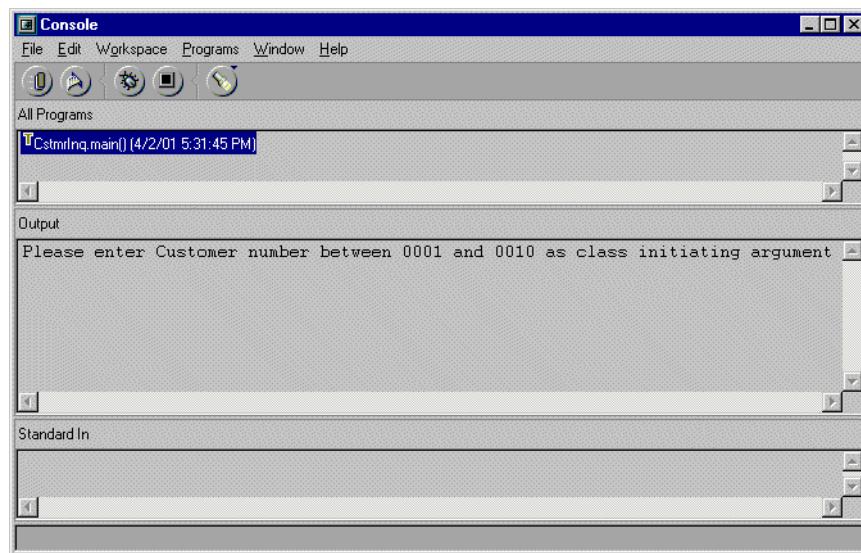


Figure 10-5 Running CstmrInq in VisualAge for Java

Now, we run the class again. This time, we give a customer number as an argument. Figure 10-6 on page 412 shows how to enter the argument.

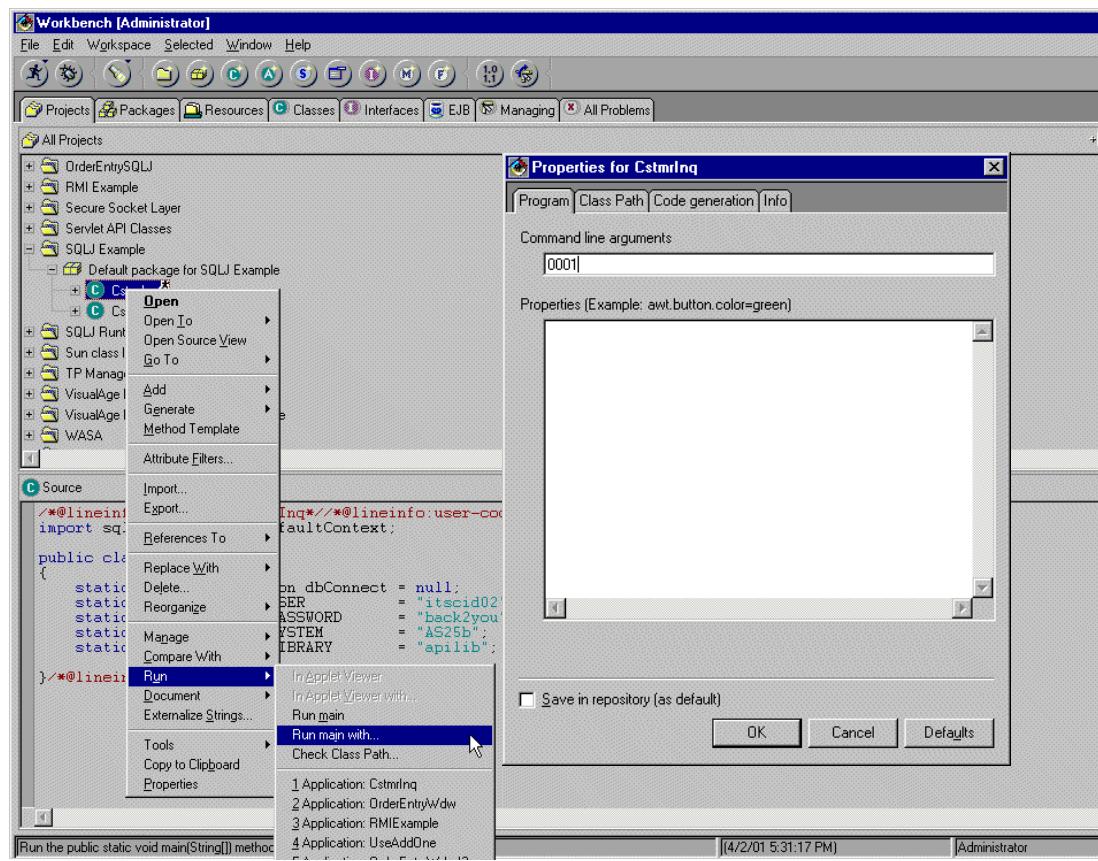


Figure 10-6 Entering the argument

The first name of the customer is displayed as shown in Figure 10-7.

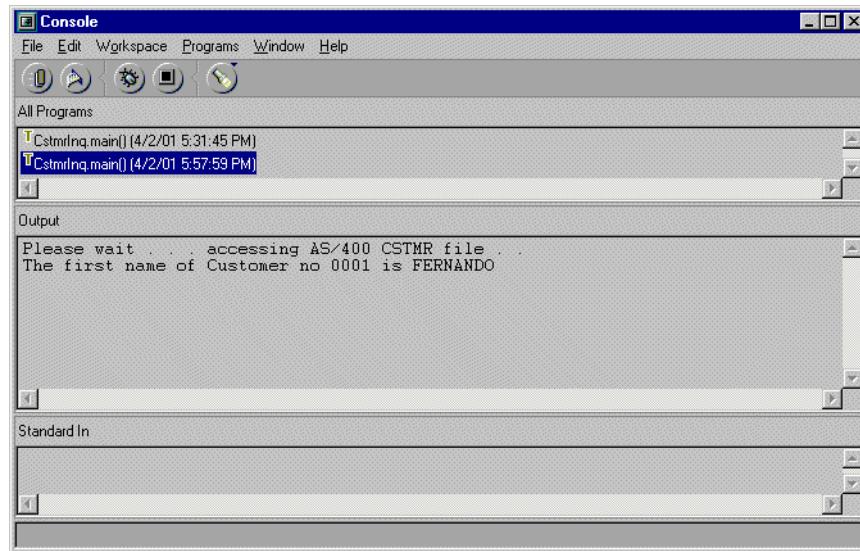


Figure 10-7 Displaying the result

### Running CstmrInq on an iSeries server

To run this example on the iSeries server, we have to export the classes and resource file to the iSeries server file system as shown in Figure 10-8.

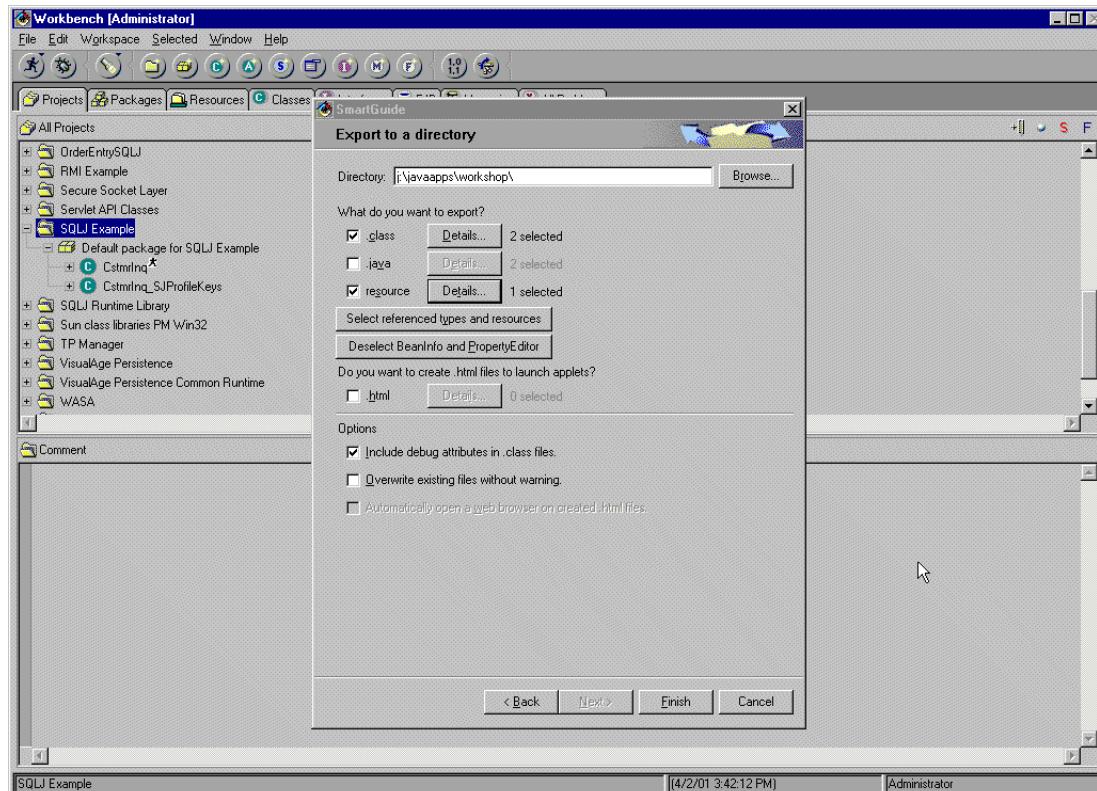


Figure 10-8 Exporting the classes and resource file

To run the program in the Qshell Interpreter, you simply enter the **java** command with the class name on the command line. For example, enter **java CstmrInq**, as shown in Figure 10-9.

```
QSH Command Entry

$ 
> java CstmrInq
Please enter Customer number between 0001 and 0010 as class initiating argument
$ 

==>

F3=Exit F6=Print F9=Retrieve F12=Disconnect
F13=Clear F17=Top F18=Bottom F21=CL command entry
```

Figure 10-9 Running CstmrInq on the iSeries server

Now, start the class again. This time give a customer number as an argument, as shown in Figure 10-10 on page 414. The first name of the customer is displayed.

```

QSH Command Entry

$ 
> java CstmrInq 0001
Please wait . . . accessing iSeries CSTMR file . .
The first name of Customer no 0001 is FERNANDO
$

====>

F3=Exit F6=Print F9=Retrieve F12=Disconnect
F13=Clear F17=Top F18=Bottom F21=CL command entry

```

*Figure 10-10 Entering a customer number*

To summarize, this Java program retrieves a single customer record at a time from iSeries CSTMR file using SQLJ and Java. It writes the first name from the record retrieved to the display.

### 10.3.3 CstmrInq explanation

Next, we explain the CstmrInq program using code snippets. Figure 10-11 shows the import statement for the package named java.sql. It is a standard Java package that is required whenever you use JDBC. The other is the import statement for the DefaultContext class in the sqlj.runtime.ref package that is required in our class.

The name of the class is CstmrInq, and it starts with the definition of a few static variables. These variables are used later in the program to provide platform specific information for system name, user ID, default library, and password.

```

import java.sql.*;
import sqlj.runtime.ref.DefaultContext;

public class CstmrInq
{
    static public String USER          = "user id";
    static public String PASSWORD      = "password";
    static public String SYSTEM        = "system name";
    static public String LIBRARY       = "library";

```

*Figure 10-11 CstmrInq class*

In the main() method that is shown in Figure 10-12, the iSeries native JDBC driver is loaded in the first try block. If we run this class from the iSeries server, the load is successful, and you can establish the connection. If the Java class runs on other platforms, the load fails and a ClassNotFoundException exception is generated. This exception is caught and handled in the catch statement. Here, another driver, the IBM Toolbox for Java JDBC driver, is loaded for other platforms.

Next, a JDBC connection is established with the iSeries server. The dbConnect object is an object that contains the JDBC connection information and is created by the static getConnection() method of DriverManager class. It is needed as a parameter to the constructor of the DefaultContext object.

We need to instantiate a `DefaultContext` object named `ctx`. It is used as a parameter for the static `setDefaultContext()` method of the `DefaultContext` class. An SQLJ application cannot run without a `DefaultContext` object.

```

public static void main (String args[]) throws java.sql.SQLException
{
    if(args.length > 0)
    {
        System.out.println("Please wait . . . accessing iSeries CSTMR file . . ");

        String url;
        try
        {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            url = "jdbc:db2://" + SYSTEM + "/" + LIBRARY;
        }
        catch(ClassNotFoundException e)
        {
            try
            {
                Class.forName("com.ibm.as400.access.AS400JDBCDriver");
                url = "jdbc:as400://" + SYSTEM + "/" + LIBRARY;
            }
            catch(ClassNotFoundException e1)
            {
                System.out.println("Jdbc driver could not be loaded " + e1);
                return;
            }
        }
        Connection dbConnect = DriverManager.getConnection(url, USER, PASSWORD);
        DefaultContext ctx = new DefaultContext(dbConnect);
        DefaultContext.setDefaultContext(ctx);
        String customerNo = args[0];
        String customerName = new String();
        #sql { select CFIRST into :customerName from CSTMR where CID = :customerNo };
        System.out.println("The first name of Customer no " + customerNo + " is " +
                           customerName);
    }
    else
    {
        System.out.println("Please enter Customer number between 0001 and 0010 "+
                           "as class initiating argument");
    }
    System.exit(0);
}

```

Figure 10-12 *Cstmrlnq main() method*

The statement shown in Figure 10-13 starts with `#sql`. This symbol denotes that it is an SQL statement that is used by the SQLJ translator. The SQLJ translator uses this statement to generate JDBC statements. It comments out this statement and inserts its own JDBC statements.

```
#sql { select CFIRST into :customerName from CSTMR where CID = :customerNo };
```

Figure 10-13 *SQLJ statement*

The statement within the {} brackets is the actual SQL statement. This is the standard ANSI static SQL. This statement reads the CSTMR table using the host variable :customerNo. It retrieves the value of the CFIRST column of the CSTMR table and stores the value in the :customerName host variable. Figure 10-14 shows the Java code that is generated by the translator. The SQLJ statement is commented out and the SQLJ runtime statements are added.

```

/*@lineinfo:generated-code///*@lineinfo:40^2*/
// ****
// #sql { select CFIRST from CSTMR where CID = :customerNo };
// *****

{
    sqlj.runtime.profile.RTResultSet __sJT_rtRs;
    sqlj.runtime.ConnectionContext __sJT_connCtx =
        sqlj.runtime.ref.DefaultContext.getDefaultContext();
    if (__sJT_connCtx == null) sqlj.runtime.error.RuntimeRefErrors.raise_NULL_CONN_CTX();
    sqlj.runtime.ExecutionContext __sJT_execCtx = __sJT_connCtx.getExecutionContext();
    if (__sJT_execCtx == null) sqlj.runtime.error.RuntimeRefErrors.raise_NULL_EXEC_CTX();
    String __sJT_1 = customerNo;
    synchronized (__sJT_execCtx) {
        sqlj.runtime.profile.RTStatement __sJT_stmt =
            __sJT_execCtx.registerStatement(__sJT_connCtx, CstmrInq_SJProfileKeys.getKey(0), 0);
        try
        {
            __sJT_stmt.setString(1, __sJT_1);
            sqlj.runtime.profile.RTResultSet __sJT_result = __sJT_execCtx.executeQuery();
            __sJT_rtRs = __sJT_result;
        }
        finally
        {
            __sJT_execCtx.releaseStatement();
        }
    }
    try
    {
        sqlj.runtime.ref.ResultSetIterImpl.checkColumns(__sJT_rtRs, 1);
        if (!__sJT_rtRs.next())
        {
            sqlj.runtime.error.RuntimeRefErrors.raise_NO_ROW_SELECT_INTO();
        }
        customerName = __sJT_rtRs.getString(1);
        if (__sJT_rtRs.next())
        {
            sqlj.runtime.error.RuntimeRefErrors.raise_MULTI_ROW_SELECT_INTO();
        }
    }
    finally
    {
        __sJT_rtRs.close();
    }
}
// ****
/*@lineinfo:user-code///*@lineinfo:40^77*/

```

Figure 10-14 Java code generated by the SQLJ translator

To summarize, this Java program uses SQLJ to retrieve data from an iSeries server table and displays it. The program can run on the iSeries server and other platforms without change.

## 10.4 The CstmrList example

The CstmrList program retrieves a group of records from the CSTMR table using SQLJ, and writes them to the display. The program runs on both the iSeries server and on a PC without change. The program performs the following tasks:

1. Sets the iterator (or cursor) to store a group of records. This is done by using an SQLJ statement.
2. Establishes a connection to the iSeries server by using the JDBC driver. If the program is running on the iSeries server, it loads the iSeries native JDBC driver. If it is running on a PC, it loads the PC IBM Toolbox for Java JDBC driver.
3. Sets the DefaultContext, which is required for using SQLJ.
4. Reads a group of records from the CSTMR table by using SQL and stores the results in the instance of the iterator (cursor), which was defined in the first step.
5. Using a while loop, each row of records that is retrieved from the iterator (cursor) is written to the display until the end of the cursor is reached.

### 10.4.1 Setting up VisualAge for Java for the CstmrList example

The setup is exactly the same as shown in 10.3.1, “Setting up VisualAge for Java for the Cstmrlnq example” on page 408.

### 10.4.2 Running CstmrList

Figure 10-15 on page 418 shows running the CstmrList class on the iSeries server, using the Qshell Interpreter. This example shows the use of the iterator (cursor) with SQLJ to retrieve a group of records from the CSTMR table.

```

QSH Command Entry

$ 
> java CstmrList
Please wait . . . accessing iSeries CSTMR file . .
Cust no 0001 is FERNANDO      ZULIANI          from PANTANAL _      BZ
Cust no 0002 is JEREK        MISZCZYK         from POLAND          TX
Cust no 0003 is Piggy        ABLE              from Denver_         CO
Cust no 0004 is TRAVIS       NELSON            from Concord_       AR
Cust no 0005 is Kareem      MULLEN-SCHULTZ   from Minneapolis_  MN
Cust no 0006 is ROBERT      MAATTA            from Damariscotta_ VT
Cust no 0007 is JIM          FAIR              from Boise_         ID
Cust no 0008 is SIMON       COULTER           from Des_Moines_   IO
Cust no 0009 is PIERRE      GOUDET            from Cheyenne_      WY
Cust no 0010 is Joe          LLAMES             from Jacksonville FL
Cust no 0011 is Warawich    Sundarabhaka     from Rochester     MN
----- End of CSTMR -----
====>
F3=Exit F6=Print F9=Retrieve F12=Disconnect
F13=Clear F17=Top F18=Bottom F21=CL command entry

```

Figure 10-15 Retrieving a group of records on iSeries server

### 10.4.3 CstmrList explanation

The basic explanation of SQLJ statements that was covered in the previous CstmrInq example in 10.3.3, “CstmrInq explanation” on page 414, also applies to this program. In this example, a group of records is retrieved. We only explain the statements that are unique.

As shown in Figure 10-16, in the beginning of the class, before the declaration of the class itself, an SQLJ statement is added. This statement is the declaration of the iterator or cursor. The SQLJ translator converts this statement into a declaration of a class of the same name as the iterator name. For this example, the CustCursor class is created. This generated class stores a group of records. You do not need to understand this generated code. You only need to know how to define the fields that you want to use in this class as a group of records in an SQLJ statement.

```

import java.sql.*;
import sqlj.runtime.ref.DefaultContext;
#sql iterator CustCursor (String id, String firstname, String lastname,
                           String city, String state);
public class CstmrList
{
    static public String USER          = "user id";
    static public String PASSWORD      = "password";
    static public String SYSTEM        = "system name";
    static public String LIBRARY       = "library";
}

```

Figure 10-16 CstmrList class

Figure 10-17 shows the main method that controls the execution of the program. First, it displays a message, which says: Please wait. This is displayed, because the initial JDBC connection takes some time and you need to inform the user of this. Then, the methods are run to establish the JDBC connection and to set the DefaultContext.

```

public static void main (String args[]) throws SQLException
{
    System.out.println("Please wait . . . accessing iSeries CSTMR file . . ");
    String url;
    try
    {
        Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        url = "jdbc:db2://" + SYSTEM + "/" + LIBRARY;
    }
    catch(ClassNotFoundException e)
    {
        try
        {
            Class.forName("com.ibm.as400.access.AS400JDBCDriver");
            url = "jdbc:as400://" + SYSTEM + "/" + LIBRARY;
        }
        catch(ClassNotFoundException e1)
        {
            System.out.println("Jdbc driver could not be loaded " + e1);
            return;
        }
    }
    Connection dbConnect = DriverManager.getConnection(url, USER, PASSWORD);
    DefaultContext ctx = new DefaultContext(dbConnect);
    DefaultContext.setDefaultContext(ctx);

    CustCursor cCursor;

    #sql cCursor = { Select CID      as "id",
                    CFIRST   as "firstname",
                    CLAST    as "lastname",
                    CCITY    as "city",
                    CSTATE   as "state"
                    from CSTMR };
    while (cCursor.next())
    {
        System.out.println( " Cust no " + cCursor.id() + " is " +
                           cCursor.firstname() + cCursor.lastname() + " from " +
                           cCursor.city() + cCursor.state() );
        System.out.println("----- End of CSTMR
-----");

        System.exit(0);
    }
}

```

*Figure 10-17 CstmrList main() method*

Each of the SQLJ related statements in the main method are explained individually in the following text.

The code shown in Figure 10-18 declares an object based on the CustCursor class. This is the same class that is created by the SQLJ translator because of the definition of the iterator at the beginning of this class.

```
CustCursor cCursor;
```

*Figure 10-18 Creating an instance of the CustCursor class*

In Figure 10-19, the select statement is issued against the instance of the iterator class that was created in the previous step. It loads the cursor with the group of records that are returned by the select statement.

**Note:** In this SQL statement, the column name of the table is linked with a literal name. These literal names have to be the same as defined at the beginning of the class definition in the iterator class. SQLJ uses these names to create getter methods for these data members.

```
#sql cCursor = { Select CID      as "id",
                  CFIRST   as "firstname",
                  CLAST    as "lastname",
                  CCITY    as "city",
                  CSTATE   as "state"
              from CSTMR };
```

Figure 10-19 Loading the cursor with a group of records

A while loop is used with the instance of the iterator class to fetch all of the records from this cursor, as shown in Figure 10-20.

**Note:** You can use the getter methods to read the data members. These methods were automatically created by SQLJ.

```
while (cCursor.next())
{
    System.out.println( " Cust no " + cCursor.id() + " is " +
                        cCursor.firstname() + cCursor.lastname() + " from " +
                        cCursor.city() + cCursor.state() );
}
```

Figure 10-20 Retrieving all of the records

Finally, the main program ends with the `System.exit` statement. This statement releases the iSeries connection and the thread. Otherwise, it is possible for the Java class not to end and just keep waiting because of the active JDBC connection. Figure 10-21 shows the Java code that is generated by the translator. The SQLJ statement is commented out and the SQLJ runtime statements are added.

```

/*@lineinfo:generated-code///*@lineinfo:41^2*/

// ****
// #sql cCursor = { Select CID as "id",
//                   CFIRST as "firstname",
//                   CLAST as "lastname",
//                   CCITY as "city",
//                   CSTATE as "state"
//                   from CSTMR };
// *****

{
    sqlj.runtime.ConnectionContext __sJT_connCtx =
    sqlj.runtime.ref.DefaultContext.getDefaultContext();
    if (__sJT_connCtx == null) sqlj.runtime.error.RuntimeRefErrors.raise_NULL_CONN_CTX();
    sqlj.runtime.ExecutionContext __sJT_execCtx = __sJT_connCtx.getExecutionContext();
    if (__sJT_execCtx == null) sqlj.runtime.error.RuntimeRefErrors.raise_NULL_EXEC_CTX();
    synchronized (__sJT_execCtx) {
        sqlj.runtime.profile.RTStatement __sJT_stmt =
        __sJT_execCtx.registerStatement(__sJT_connCtx, CstmrList_SJProfileKeys.getKey(0), 0);
        try {
            sqlj.runtime.profile.RTResultSet __sJT_result = __sJT_execCtx.executeQuery();
            cCursor = new CustCursor(__sJT_result);
        }
        finally {
            __sJT_execCtx.releaseStatement();
        }
    }
}

// *****

/*@lineinfo:user-code///*@lineinfo:46^40*/

```

*Figure 10-21 Java code generated by the SQLJ translator*

To summarize, in this example, the following actions occurred:

- ▶ Defined an iterator
- ▶ Established a JDBC connection
- ▶ Set the DefaultContext class
- ▶ Loaded a group of records into the instance of the iterator using SQLJ
- ▶ Using the generated getter methods, the data members were fetched from the iterator and written to the display

## 10.5 Order Entry client application

This section uses the client side of the Order Entry application that was discussed Chapter 8, “Migrating the user interface to the Java client” on page 321, and converts the use of JDBC stored procedures to use SQLJ. In addition, we convert access to the ITEM file from DDM record-level access to SQLJ. This involves changing three programs:

- ▶ OrderEntryWdw
- ▶ SltCustWdw
- ▶ SltItemWdw

Figure 10-22 shows the new application design.

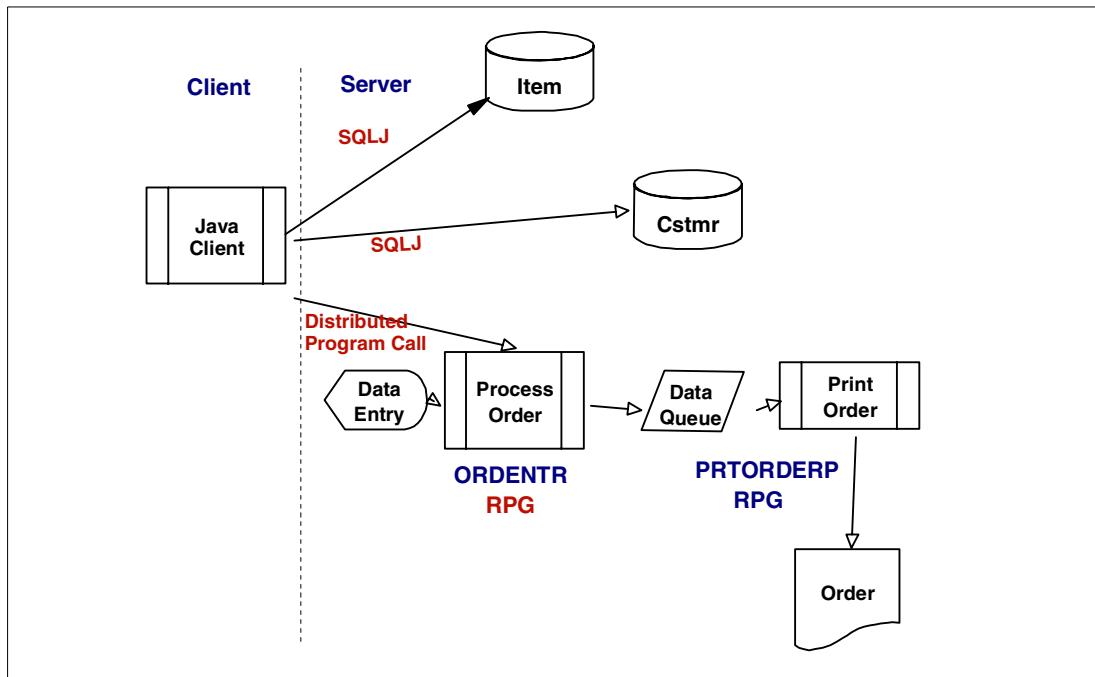


Figure 10-22 Order Entry client using SQLJ

Figure 10-23 shows the main Order Entry window and the interfaces used.

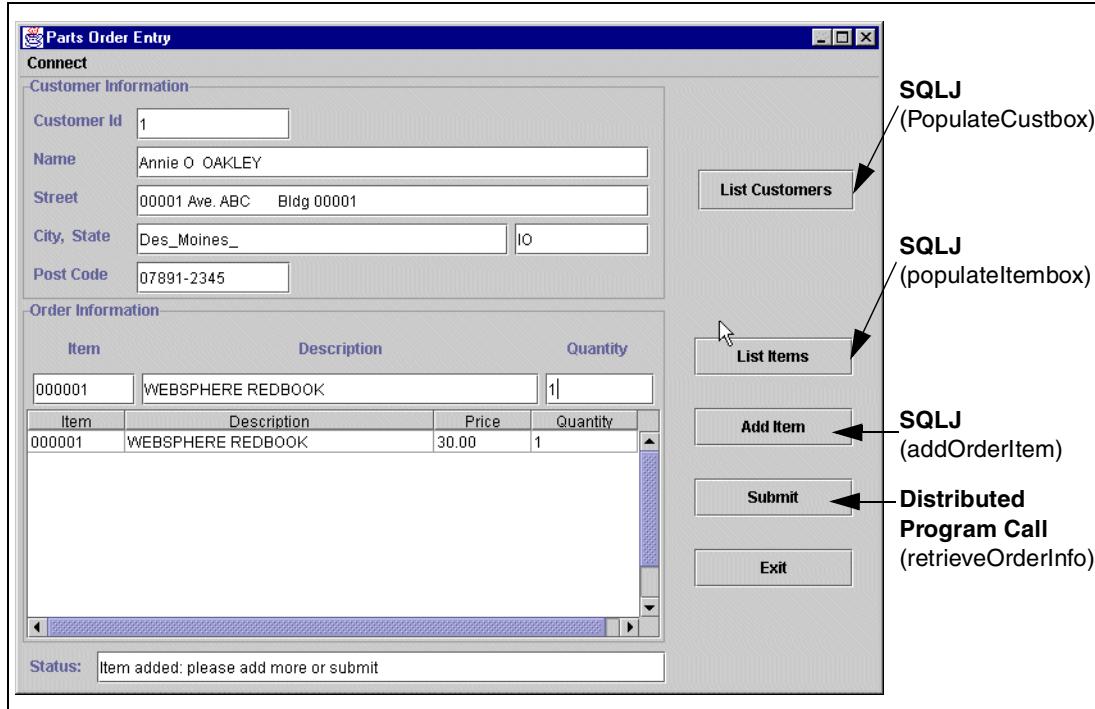


Figure 10-23 Order Entry client interfaces

## 10.5.1 Converting the OrderEntryWdw class to use SQLJ

This section shows the code that we add, change, and remove to use SQLJ in this application.

### Modifying the OrderEntryWdw class

We need to add a statement in the `connectToDB()` method to set up the `DefaultContext` class with the instance of the `DefaultContext` class. The `Connection` object named `dbConnect` is the parameter of the constructor of the `DefaultContext` object. Since we are replacing the DDM code, we have no need to run the `openItemfile()` method anymore, so we comment it out. This code is shown in Figure 10-24.

```
private boolean connectToDB(String systemName, String userid, String password)
{
    updateStatus("Connecting to " + systemName + " ...");
    this.systemName = systemName;
    this.userid = userid;
    this.password = password;
// openItemFile();
    try
    {
        DriverManager.registerDriver(new com.ibm.as400.access.AS400JDBCDriver());
        dbConnect = DriverManager.getConnection("jdbc:as400://" + systemName +
            "/apilib;naming=sql;errors=full;date format=iso;" +
            "extended dynamic=true;package=JavaMig;package library=apilib",
            userid, password);
        // addition for SQLJ
        sqlj.runtime.ref.DefaultContext.setDefaultContext(
            new sqlj.runtime.ref.DefaultContext(dbConnect));
    }
    catch (Exception e)
    {
        updateStatus("Connect failed");
        handleException(e);
        return false;
    }

    updateStatus("Connected to " + systemName);
    return true;
}
```

Figure 10-24 Modifying the `connectToDB()` method

The other method that we have to modify is the `addOrderItem()` method. This method reads the ITEM file to verify that an item being ordered is valid. Figure 10-25 on page 424 shows how we use SQLJ to validate an item.

```

private void addOrderItem(String key)
{
    // This method is invoked when the add item button
    // is pressed. It gets the text from the item text
    // field (getItemTF().getText()) and uses it as a
    // key to read the ITEM file.

    String[] orderRow = new String[4];
    String oid      = new String();
    String oname    = new String();
    String oprice   = new String();
    String odata    = new String();
    String oqty     = getQtyTF().getText();

    try
    {
        #sql { select IID, INAME, IPRICE into :oid, :oname, :oprice
               from ITEM where :key = IID };
    }
    catch(SQLException e)
    {
        updateStatus("SQL Exception: Error retrieving field data from Item file");
        handleException(e);
        return;
    }

    orderRow[0]= oid;
    orderRow[1]= oname;
    orderRow[2]= oprice;
    orderRow[3]= oqty;
    getDefaultTableModel1().addRow(orderRow);

    getSubmitBTN().setEnabled(true);
    updateStatus("Item added: please add more or submit");
}

```

*Figure 10-25 Modifying the addOrderItem() method*

In the statements that are shown in Figure 10-25, an SQL statement is used to retrieve a record from the ITEM file based on the value of the key. The key value is based on the item selected from the display. You need to define the host variables where SQLJ returns the retrieved values.

In SQLJ statements, you cannot directly use expressions and arrays. Instead, simple Strings are required. This is the reason retrieved variables are first loaded into simple String objects and then these Strings are loaded into array cells. The array is used as a row to populate the DefaultTableModel object for display.

### 10.5.2 Converting SltCustWdw to use SQLJ

In this section, we convert the SltCustWdw class from using JDBC stored procedures to access the CSTMR table using SQLJ. As shown in Figure 10-26, this program is used to display a list of customers to the end user.



Figure 10-26 SltCustWdw window

### Modifying the SltCustWdw class

Figure 10-27 defines the iterator or the cursor CustList class. For a detailed explanation, see the CstmrList class example in 10.4.3, “CstmrList explanation” on page 418.

```
#sql iterator CustList (String id,
    String last,
    String first,
    String init,
    String addr1,
    String addr2,
    String city,
    String state,
    String zip);

public class SltCustWdw extends javax.swing.JFrame implements
java.awt.event.ActionListener, java.awt.event.WindowListener {
```

Figure 10-27 Defining the iterator

We modify the populateCustBox() method as shown in Figure 10-28 on page 426. An instance of the iterator CustList class is created. A group of records from the CSTMR file is loaded into the cList object. By using a while loop and getter methods, the data members are retrieved from the cList object and loaded into the DefaultTableModel object for display.

```

private void populateCustBox() {
    orderWindow.updateStatus("Retrieving customer list...");

    // The result set that is returned represents records that
    // have 9 fields of data. These fields are all character
    // data and will be stored in an array of strings

    try
    {
        CustList cList;

        #sql cList = {select    CID      as "id",
                      CLAST    as "last",
                      CFIRST   as "first",
                      CINIT    as "init",
                      CADDR1   as "addr1",
                      CADDR2   as "addr2",
                      CCITY    as "city",
                      CSTATE   as "state",
                      CZIP     as "zip"
                  from CSTMR for read only};

        while (cList.next())
        {
            String[] array= new String[9];
            array[0]=cList.id();
            array[1]=cList.last();
            array[2]=cList.first();
            array[3]=cList.init();
            array[4]=cList.addr1();
            array[5]=cList.addr2();
            array[6]=cList.city();
            array[7]=cList.state();
            array[8]=cList.zip();
            getDefaultTableModel().addRow(array);
        }
    }
    catch (SQLException e)
    {
        orderWindow.updateStatus("Error retrieving customer list");
        handleException(e);
        return;
    }
    orderWindow.updateStatus("Customer list retrieved");
    return;
}

```

*Figure 10-28 Modifying the populateCustBox() method*

It is not necessary to set the DefaultContext class because it was already done by OrderEntryWdw. In this example, the new class uses SQLJ to retrieve a group of records from the iSeries server instead of the JDBC stored procedure. The original code can be reviewed in “private void populateCustBox()” on page 330.

### 10.5.3 Converting SltItemWdw to use SQLJ

In this section, we convert the SltItemWdw program from using JDBC stored procedures to access the ITEM table to use SQLJ. As shown in Figure 10-29, this program is used to display a list of items to the end user.



Part #	Description	Price	Quantity
000001	WEBSPHERE REDBOOK	\$30.00	317
000002	Radio_Controlled_Plane	\$96.86	7
000003	Change_Machine	\$52.55	37
000004	Baseball_Tickets	\$91.14	899
000005	Twelve_Nurn_Two_Pencils	\$50.58	1715
000006	Over_Under_Shotgun	\$79.66	1306
000007	Feel_Good_Vitamins	\$29.81	200
000008	Cross_Country_Ski_Set	\$93.00	54
000009	Rubber_Baby_Buggy_Wheel	\$98.71	200
000010	ITSO REDBOOK SG24-2152	\$50.00	297
000011	ITSO REDBOOK SG24-2163	\$20.87	982
000012	Plastic_Garbage_Pail	\$22.40	84
000013	Doll_House_Furniture	\$24.22	541

Figure 10-29 SltItemWdw window

### Modifying the SltItemWdw class

Figure 10-30 defines the iterator or the cursor class ItemList. For a detailed explanation, see the CstmrList class example in 10.4.3, “CstmrList explanation” on page 418.

```
#sql iterator ItemList (String ijid,
    String iname,
    String iprice,
    String iqty);

public class SltItemWdw extends javax.swing.JFrame implements
    java.awt.event.ActionListener, java.awt.event.WindowListener {
```

Figure 10-30 Defining the iterator

We modify the populateItemBox() method as shown in Figure 10-31 on page 428. The instance of the iterator class ItemList is created. Then, a group of records from the ITEM file and STOCK file are read by joining these files using IID and STIID as keys and loading them into iList. By using a while loop and getter methods, we retrieve the data members from iList object and load them into the DefaultTableModel object for display.

```

private void populateItemBox() {
    orderWindow.updateStatus("Retrieving item list...");

    try
    {
        ItemList iList;

        #sql iList = {Select IID      as "ijid",
                      INAME   as "iname",
                      IPRICE as "iprice",
                      STQTY   as "iqty"
                  from ITEM, STOCK
                  where IID = STIID and STWID = '0001'
                  for fetch only };

        while (iList.next())
        {
            String[] array=new String[4];
            array[0]=iList.ijid();
            array[1]=iList.iname();
            array[2]="$" + iList.iprice().toString();
            array[3]=iList.iqty().toString();
            getDefaultTableModel1().addRow(array);
        }
    }
    catch (SQLException e)
    {
        orderWindow.updateStatus("Error retrieving item list");
        handleException(e);
        return;
    }
    orderWindow.updateStatus("Item list retrieved");
    return;
}

```

*Figure 10-31 Modifying the populateItemBox() method*

It is not necessary to set the DefaultContext class because it was already done by OrderEntryWdw. In this example, the new class uses SQLJ to retrieve a group of records from the iSeries server instead of the JDBC stored procedure. The original code can be reviewed in “private void populateItemBox()” on page 335.

## 10.6 Order Entry server application

In this section, we convert the Order Entry server side application discussed in Chapter 9, “Moving the server application to Java” on page 349, from using JDBC to use SQLJ. All of the statements that you need to add, remove, or change are documented below. In this example, we use different types of SQL statements, such as SELECT, INSERT, and UPDATE.

Figure 10-32 shows the new application design.

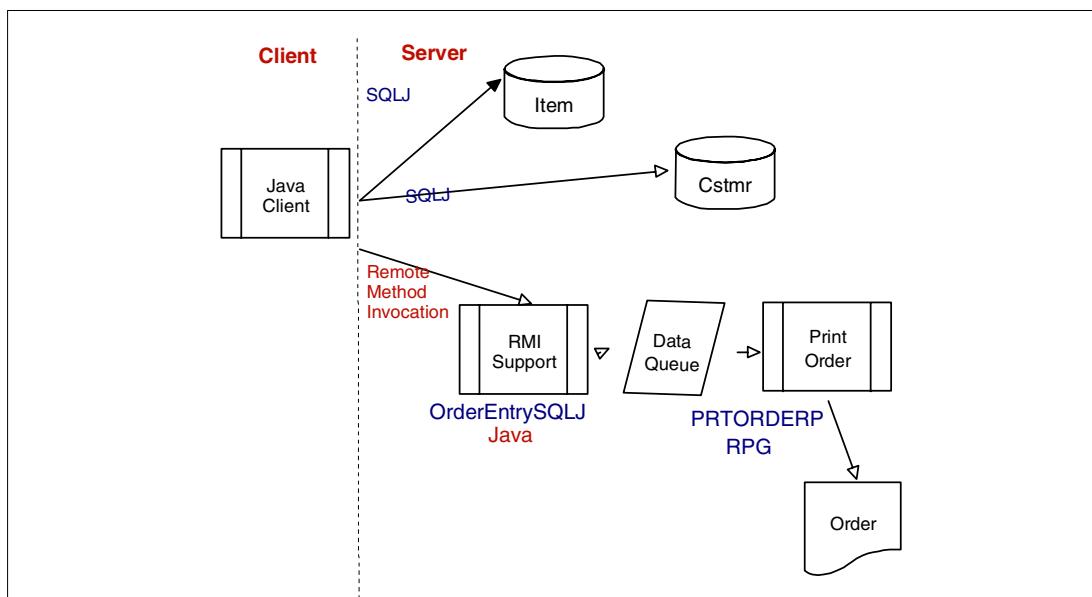


Figure 10-32 Order Entry host SQLJ application

Figure 10-33 shows the main Order Entry window and the interfaces used.

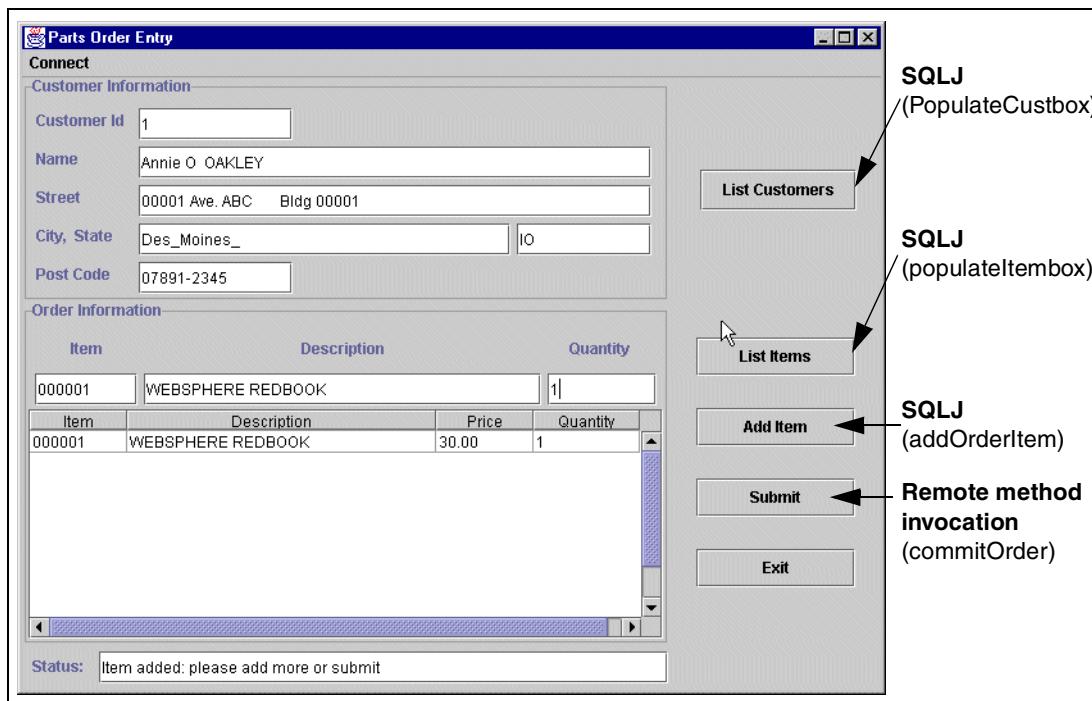


Figure 10-33 Order Entry application interface

### 10.6.1 Setting up the Order Entry server application

The SQLJ application is based on the JDBC application named OrderEntryJDBC found in the serverOrderEntry package. We export the OrderEntryJDBC source code from VisualAge for Java into the file system directory. Then, we change the file and the class name to OrderEntrySQLJ and use .sqlj as the file extension.

This section shows you which statements you must add to use SQLJ and which JDBC statements you need to remove from the .sqlj file.

Since we cannot add embedded SQLJ statements directly to our source code in a source pane, we edit the program source using a Windows 95/NT editor. After the modification, we import the .sqlj file into our project resource directory. The Java classes and resource file will be generated by VisualAge for Java. The OrderEntrySQLJ class will be placed in serverOrderEntry package. The class and its methods are shown in Figure 10-34.

```
serverOrderEntry Package
  OrderEntrySQLJ Class
    commitOrder()
    addOrderLine()
    addOrderHeader()
    updateStock()
    updateCustomer()
    writeDataQueue()
    getOrderNumber()
    finalize()
    initialize()
```

Figure 10-34 OrderEntrySQLJ class

### 10.6.2 Converting the Order Entry to use SQLJ

This section discusses the changes required to the host program to use SQLJ instead of JDBC for database access.

Figure 10-35 shows the class definition for the SQLJ version. We remove some of the class variables used by the JDBC version.

```

import java.math.*; // for BigDecimal class
import java.sql.*; // for JDBC classes
import java.util.*; // for Properties class
import java.text.*; // for DateFormat class
import java.rmi.*; // for Remote Method Invocation
import java.rmi.server.*;
import orderObjects.*;

/**
 * This class was generated by a SmartGuide.
 *
 * This class is a replacement for the ORDENTR RPG IV program.
 * The method and variable names have been improved slightly
 * since Java supports longer names than RPG.
 */
public class OrderEntrySQLJ extends UnicastRemoteObject implements OrderEntryI
{
    // Mnemonic values
    private static final String SYSTEM_LIBRARY = "QSYS.LIB";
    private static final String DATA_QUEUE_NAME = "ORDERS.DTAQ";
    private static final String DATA_QUEUE_LIBRARY = "APILIB.LIB";
    private static final String WAREHOUSE = "0001";
    private static final int DISTRICT = 1;
    private static final String SYSTEM = "localhost"; // enter system name here
    private static final String USER = "*current";
    private static final String PASSWORD = "*current";
    private static final String DATA_LIBRARY = "APILIB";

    // an AS400 object
    private AS400 as400 = null;

    // A global connection and prepared statement
    private Connection dbConnection = null;
}

```

*Figure 10-35 The class definition of OrderEntrySQLJ*

### The initialize() method

The addition is to execute the static setDefaultContext() method of the DefaultContext class. The parameter of the method is a new instance of its class. The constructor of this instance uses the JDBC connection object as a parameter. The JDBC connection, dbConnect, is used to connect to database on iSeries server.

The as400 object is still used when we instantiate IBM Toolbox for Java objects in the writeDataQueue() method. We remove the statements used by the JDBC version.

```

private void initialize () throws Exception
{
    System.out.println("initialize:new");

    // Connect to the iSeries
    as400 = new AS400(SYSTEM, USER, PASSWORD);

    // Create a properties object for JDBC connection
    Properties jdbcProperties = new Properties();

    // Set the properties for the JDBC connection
    jdbcProperties.put("user", USER);
    jdbcProperties.put("password", PASSWORD);
    jdbcProperties.put("naming", "sql");
    jdbcProperties.put("errors", "full");
    jdbcProperties.put("date format", "iso");

    // Load the AS400 Native JDBC driver into the JVM
    // This method automatically verifies the existence of the driver
    // and loads it into the JVM -- should not use DriverManager.registerDriver()
    Class.forName ("com.ibm.db2.jdbc.app.DB2Driver");
    // DriverManager.registerDriver(new AS400JDBCDriver());

    // Connect using the properties object
    dbConnection = DriverManager.getConnection("jdbc:db2:"+SYSTEM+"/"+DATA_LIBRARY,
    jdbcProperties);

    // Addition for SQLJ
    sqlj.runtime.ref.DefaultContext.setDefaultContext(
        new sqlj.runtime.ref.DefaultContext(dbConnection));

    return;
}

```

*Figure 10-36 The initialize() method*

### The commitOrder() method

No changes are required to this method.

### The addOrderHeader() method

We replace the JDBC statements with the SQLJ statement that is used to insert an order row into the ORDERS table as shown in Figure 10-37.

```
private void addOrderHeader (String aCustomerNumber,
                           BigDecimal anOrderNumber,
                           BigDecimal anOrderLineCount) throws Exception
{
    // Get the current date and time
    java.util.Date currentDateTime = new java.util.Date();

    String cDate = getRawDate(currentDateTime);
    String cTime = getRawTime(currentDateTime);

    #sql { INSERT INTO ORDERS (OWID, ODID, OCID, OID, OLINES, OCARID, OLOCAL, OENTDT,
                               OENTTM)
           VALUES(:WAREHOUSE, :DISTRICT, :aCustomerNumber, :anOrderNumber,
                  :anOrderLineCount, 'ZZ', 1, :cDate, :cTime ) };
    return;
}
```

Figure 10-37 The `addOrderHeader()` method

### The `addOrderLine()` method

We replace the JDBC statements with the SQLJ statement that is used to insert an order row into the ORDLIN table as shown in Figure 10-38 on page 434.

```

private BigDecimal addOrderLine (BigDecimal anOrderNumber,
                               Order anOrder) throws Exception
{
    BigDecimal orderTotal = new BigDecimal(0);

    int lineCounter = 0;

    // Get the customer discount percentage
    BigDecimal customerDiscount = getCustomerDiscount(anOrder.getCustomerId());

    // Get Order detail elements from the vector
    java.util.Enumeration e = anOrder.getOrderDetail().elements();

    // While we have order lines to process
    while(e.hasMoreElements())
    {
        OrderDetail orderLine = (OrderDetail) e.nextElement();

        BigDecimal blineCounter = new BigDecimal(++lineCounter);
        String bgetItemId = orderLine.getItemId();
        BigDecimal bgetItemQty = orderLine.getItemQty();
        BigDecimal orderAmount = (orderLine.getItemPrice().subtract(
            orderLine.getItemPrice().multiply(customerDiscount).
            divide(new BigDecimal(100),BigDecimal.ROUND_DOWN))).
            multiply(orderLine.getItemQty()).setScale(2, BigDecimal.ROUND_HALF_UP);

        #sql { INSERT INTO ORDLIN (OLOID, OL DID, OLWID, OLNBR, OLSPWH, OLIID, OLQTY,
                                     OLAMNT, OL DLVD, OL DLVT)
               VALUES (:anOrderNumber,:DISTRICT,:WAREHOUSE,:blineCounter,'JAVA',
                      :bgetItemId,:bgetItemQty,:orderAmount, 12311999, 235959) };

        // Accumulate the order total
        orderTotal = orderTotal.add(orderAmount);

        // Update the stock record
        updateStock(orderLine.getItemId(), orderLine.getItemQty());
    }
    return orderTotal;
}

```

*Figure 10-38 The addOrderLine() method*

### The getCustomerDiscount() method

We replace the JDBC statements with the SQLJ statement that is used to retrieve the customer discount from the CSTMR table as shown in Figure 10-39.

```

private BigDecimal getCustomerDiscount (String aCustomerID) throws Exception
{
    BigDecimal customerDiscount = null;
    #sql { select CDCT into :customerDiscount from CSTMR where CID = :aCustomerID };

    return customerDiscount;
}

```

*Figure 10-39 The getCustomerDiscount() method*

## The getOrderNumber() method

We replace the JDBC statements with SQLJ statements. The row is retrieved from the DSTRCT table and is updated with the order number column, DNXTOR. It is increased by one.

```
private BigDecimal getOrderNumber () throws Exception
{
    // Get the next available order number
    BigDecimal orderNumber = null;
    #sql { select DNXTOR into :orderNumber from DSTRCT where DID = :DISTRICT and
           DWID = :WAREHOUSE };
    // Update the order number (positioned update)
    BigDecimal aorderNumber = orderNumber.add(new BigDecimal(1));
    #sql { update DSTRCT set DNXTOR = :aorderNumber where DID = :DISTRICT and
           DWID = :WAREHOUSE };
    return orderNumber;
}
```

Figure 10-40 The getOrderNumber() method

## The updateCustomer() method

We replace the JDBC statements with SQLJ statements. The row is retrieved from the CSTMR table and is updated with the new values of the columns CLDATE, CLTIME, CBAL, CYTD.

```
private void updateCustomer (String aCustomerID,
                            BigDecimal anOrderTotal ) throws Exception
{
    java.util.Date currentDateTime = new java.util.Date();

    String      uldate = null;
    BigDecimal ubal   = null;
    BigDecimal uytdd  = null;
    String      ultime = null;

    #sql { select CLDATE, CBAL, CYTD, CLTIME into :uldate, :ubal, :uytdd, :ultime
            from CSTMR where CID = :aCustomerID };

    uldate = getRawDate(currentDateTime);
    ultime = getRawTime(currentDateTime);
    ubal   = ubal.add(anOrderTotal);
    uytdd  = uytdd.add(anOrderTotal);

    #sql { update CSTMR set CLDATE = :uldate, CLTIME = :ultime, CBAL = :ubal,
           CYTD = :uytdd
           where CID = :aCustomerID };

    return;
}
```

Figure 10-41 The updateCustomer() method

## The updateStock() method

We replace the JDBC statements with SQLJ statements. The row is retrieved from the STOCK table and is updated with the new value of the column STQTY.

```

private void updateStock (String aPartNbr,
                        BigDecimal aPartQty ) throws Exception
{
    BigDecimal ustqty = null;
    #sql { select STQTY into :ustqty from STOCK where STIID = :aPartNbr
            and STWID = :WAREHOUSE };

    ustqty = ustqty.subtract(aPartQty);
    #sql { update stock set STQTY = :ustqty where STIID = :aPartNbr
            and STWID = :WAREHOUSE };

    return;
}

```

*Figure 10-42 The updateStock() method*

### **The writeDataQueue() method**

No changes are required to this method.

### **The getRawDate() method**

No changes are required to this method.

### **The getRawTime() method**

No changes are required to this method.

### **The finalize() method**

The JDBC close() method statements are removed.

```

protected void finalize( ) throws Throwable
{
    // Close the AS400 connection
    if (!dbConnection.isClosed())
    {
        dbConnection.close();
        dbConnection = null;
    }

    // Close the AS400 object
    if (as400.isConnected())
    {
        as400.disconnectAllServices();
        as400 = null;
    }

    super.finalize();

    return;
}

```

*Figure 10-43 The finalize() method*

In summary, we changed the Order Entry application to be entirely SQLJ based. The client programs were converted from using JDBC stored procedures and DDM record-level access to use SQLJ. The host program was converted from using JDBC to using SQLJ.

## 10.7 SQLJ considerations

You can edit an SQLJ file using a workstation editor and import it into the VisualAge for Java project resource directory. You can also directly edit it from the Resource page of the Project browser. Keep in mind that it is not automatically translated into Java code. The synchronization of the SQLJ file and Java code is a manual process. You should synchronize the SQLJ file with the Java source every time you edit the SQLJ file.





# Java Native Interface (JNI)

This chapter explains the integration of Java methods using the Java Native Interface (JNI) with C and RPG on the iSeries server. We use these examples:

- ▶ RPG native method
- ▶ Calling Java method from RPG
- ▶ RPG Order Entry application (six RPG procedures)
- ▶ Hello C
- ▶ Change Java String Object from C
- ▶ C with Java Invocation API

This chapter teaches you about JNI through programming examples. In addition, you learn how to set up your iSeries server to use JNI. Only minimal attention is given to the general theory of JNI.

This chapter is a starting point for using JNI with C and RPG on the iSeries server. It is intended for Java programmers who want to use the Java Native Interface to call RPG or C programs or vice versa. We assume that you have a basic understanding of Java and object oriented programming.

## 11.1 Introduction to Java Native Interface

JNI is the native programming interface for Java that is part of the Java Development Kit (JDK). JNI allows Java programs that run within a Java virtual machine (JVM) to operate with applications and libraries that are written in other languages, such as C, C++, and RPG. In addition, the Invocation API allows you to embed the JVM into your native applications.

You use JNI to write native methods to handle those situations when an application cannot be written entirely in the Java programming language. For example, you may need to use native methods and JNI in these situations:

- ▶ The standard Java class library does not support the platform-dependent features that your application needs.
- ▶ You have a library or application that is written in another programming language and you want to make it accessible to Java applications.
- ▶ You want to implement a small portion of time-critical code in a lower-level programming language, such as C, and have your Java application call these functions.

Programming with the JNI framework lets you use native methods to perform many operations. You may use native methods to represent legacy applications or explicitly to solve a problem that is best handled outside of the Java programming environment. The JNI framework lets your native method use Java objects in the same way that Java code uses these objects.

A native method can create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects that are created by Java application code. A native method can even update Java objects that it created or that were passed to it. These updated objects are available to the Java application. Therefore, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them.

Native methods can also call Java methods. Often you will already have developed a library of Java methods. Your native method does not need to “re-invent the wheel” to perform functionality that is already incorporated in existing Java methods. The native method, using the JNI framework, can call the existing Java method, pass it the required parameters, and get the results back when the method completes.

### 11.1.1 JNI positioning

JNI, which is new in JDK 1.1, takes the place of the Native Method Interface in JDK 1.0. Netscape offered the Java Runtime Interface (JRI) as a comprehensive environment for the Netscape JVM. While all of the existing native interfaces may have their own various strengths and weaknesses, the Netscape JRI was used as a starting point for JNI. Now, JNI is the JDK standard.

The primary goal of JNI is to provide a standard way to interface with programs written in other languages. This allows you to maintain a single version of your native method libraries on that platform.

JNI has two main purposes:

- ▶ It specifies a way to write Java native methods. Java native methods invoke code written in other languages native to the platform on which Java is running.

- ▶ It includes an Invocation API for embedding a JVM in native applications. The JNI standard may give a native library its “best chance” to run in a given JVM, according to JavaSoft.

The JNI may not be the only native method interface that a given Java VM supports. The benefit of JNI is that it is a standard part of Java. And as a standard interface, this benefits programmers who want to load their native code libraries into different JVMs. In some special cases, you may have to use a lower-level, VM-specific interface to achieve top efficiency or you may use a higher-level interface to build software components.

To summarize, JNI offers the option of using a standard method of integrating a Java method with procedures from other languages such as C, C++, and RPG. It is easy to see that the JNI serves as the glue between Java and native applications. Figure 11-1 shows how the JNI ties the C side of an application to the Java side.

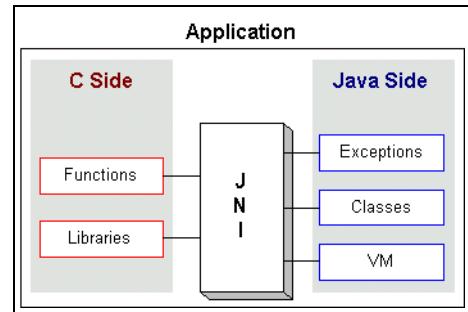


Figure 11-1 JNI definition

### 11.1.2 Who should use JNI

C, C++, or RPG programmers who want to integrate with Java methods may want to start programming in JNI. JNI protects your code from unknowns, such as the vendor-specific VM extension that is used to integrate traditional language with Java. By conforming to the JNI standard, a native library (C, C++, or RPG) is given the best option to run problem free in any standard implementation of a JVM.

JavaSoft has tried to ensure that the JNI does not impose any overhead or restrictions on any VM implementation, including object representation, garbage collection scheme, and so on. Therefore, it is an efficient method that you can use to integrate Java with traditional languages.

For the iSeries server, several other alternatives to using the Java Native Interface are available including:

- ▶ IBM Toolbox for Java access classes such as Distributed Program Call and Data Queue support
- ▶ The Program Call Markup Language (PCML) support available with the IBM Toolbox for Java Modification 2 and later

The IBM Toolbox for Java access classes and PCML are easier to program than JNI. The advantage of using JNI is that both the calling program and the called program run in the same process (job) on the iSeries server, while the other methods start a new process (job). This makes JNI calls faster at startup time and less resource intensive.

### 11.1.3 Additional information

If you want to find out more about JNI, the best place to start is the JavaSoft Web site, which is located at: <http://www.javasoft.com/>

The Web sites that we found useful are:

- ▶ **JNI tutorial:** <http://java.sun.com/nav/read/Tutorial/native1.1/index.htm>

This URL contains:

- Overview of JNI
- Writing Java programs with native methods
- Integrating Java and native programs
- Interacting with Java from the native side
- Starting the JVM
- Summary of JNI

- ▶ **JNI documentation:**

<http://www.javasoft.com/products/jdk/1.2/docs/guide/jni/index.html>

This URL contains:

- JNI 1.1 specification
- JNI enhancements in the Java 2 SDK
- Java Native Interface tips - FAQ

## 11.2 Java Native Interface and RPG

This section covers using JNI to interface with RPG programs. You should only consider using JNI and RPG on an iSeries server at V4R4 or later. This is because *RPG is not thread safe in releases prior to V4R4* and unpredictable results may occur.

There are a number of requirements and restrictions when using RPG and JNI. We provide an overview of them here to help you better understand the programming examples:

- ▶ The RPG application code to be called must be in a *service program*. This means the RPG code must be a *true ILE*. It cannot use DFTACTGRP(\*yes), and the call is a *procedure call* rather than a program call.
- ▶ The RPG application code to be called must be prototyped. It cannot use PARM or PLIST.
- ▶ The EXTPROC keyword is used to prototype a Java method. The format is:  
`EXTPROC(*JAVA:class_name:method_name)`  
Both the class name and the method name must be character constants. The class name must be a fully qualified Java class name.
- ▶ Starting at V5R1, ILE RPG supports the object data type by specifying the **O** data type of the D-specification. The CLASS keyword is used to provide the class of the object. The format is:  
`CLASS(*JAVA:class_name)`  
The class name must be a character literal or named constant and must be fully qualified.
- ▶ The data types of the parameters and the returned value of the method have to be mapped to Java data types. The detail of mapping data types can be found in Chapter 11 of the *ILE RPG for AS/400 V5R1 Programmer's Guide*, SC09-2507.
- ▶ The native method has to be a subprocedure because of the names, but it does not necessarily have to be in a NOMAIN module. However, we do not recommend having Java call a subprocedure in a module with a main procedure, since you can get strange results calling the main procedure from a subprocedure.
- ▶ The THREAD(\*SERIALIZE) keyword should also be specified on the H specification, so that the RPG service program can safely be called from a threaded Java application.  
Since Java is fundamentally multi-threaded, there is a requirement for modules containing

RPG native methods to specify THREAD(\*SERIALIZE). This forces all native methods in a module to synchronize on a single object. This requirement is a serious impediment to the usefulness of RPG as a “general-purpose” native method language.

### 11.2.1 RPG native method example

In this example, we show how to call an RPG procedure from a Java program using JNI. This example completes the following actions:

1. A Java class, named DistrictJ is executed. It accepts a string parameter. This parameter is a District Id and must contain a value between 1 and 10.
2. The DistrictJ object passes the District Id value as an **int** to an RPG native method named **dist**, which is a procedure in a RPG program named DISTRICTR.
3. The RPG program uses the District Id value as a key to read a record from the DSTRCT file.
4. It retrieves the year-to-date (YTD) balance from the record.
5. The YTD balance is passed back to the Java program as a return value.
6. The Java program writes the District Id and the YTD balance on the display.

This example demonstrates how to establish the link between Java methods and RPG procedures. It also shows you how to exchange Java primitive types with RPG.

The complete Java program is shown in the following section in segments to fully explain each element of code.

#### The Java class: DistrictJ

This section discusses the DistrictJ Java program that calls the RPG program using JNI.

To use non-Java methods (for example, an RPG or C procedure within Java), use the *native* modifier keyword with the method declaration. The native modifier keyword denotes that this method is not defined in a Java program and it is external to the program. The rest of the method definition follows the standard Java method definition. The native method can be either a static or an instance method. In this case, it is an instance method that has an integer value as a parameter, as shown in Figure 11-2.

```
public class DistrictJ
{
    native float dist(int districtId);
```

Figure 11-2 Using a non-Java method in a Java class

The Java program needs to know in which service program the procedure is located. Figure 11-3 shows the standard way to define an RPG service program within Java. We use a static statement to execute the System.loadLibrary method to load the service program. We must have the service program in our library list.

```
static
{
    System.loadLibrary("DISTRICTR");
}
```

Figure 11-3 Loading an RPG service program in Java

Figure 11-4 on page 444 demonstrates the `main()` method. We instantiate the `DistrictJ` object and then execute its `dist()` native method. The `String` object `args[0]` that is received as keyboard input is converted into an integer, which will be passed to the RPG native method.

```
public static void main(String args[])
{
    if(args.length > 0)
    {
        DistrictJ district = new DistrictJ();
        float ytdBalance = district.dist(Integer.parseInt(args[0]));
        System.out.println("For district "+args[0]+
                           " YTD Balance retrieved by RPG JNI program = "+
                           ytdBalance);
    }
    else
        System.out.println("Please enter DistrictId between 1 and 10");
}
```

Figure 11-4 Calling the RPG native method in the `main()` method

**Note:** In this method call, the name of the Java class is given. However, the method that executes is a native method and actually an RPG procedure.

A float variable named `ytdBalance` is returned by the RPG program. It is displayed as shown in Figure 11-5.

```
QSH Command Entry

$ 
> java DistrictJ 0001
  For district 0001 YTD Balance retrieved by RPG JNI program = 31573.2
$ 

====>

F3=Exit F6=Print F9=Retrieve F12=Disconnect
F13=Clear F17=Top F18=Bottom F21=CL command entry
```

Figure 11-5 Displaying the YTD balance

This completes the Java program. Next, the corresponding RPG program is explained.

### The RPG service program: DISTRICTR

The RPG procedure that is called from Java must be located in a service program. The modules that go into a service program often do not have a main procedure. We take advantage of the `NOMAIN` keyword on the `H` specification to tell the RPG compiler to omit the RPG cycle logic from the compiled object. This improves the performance of invoking this RPG procedure.

The `THREAD(*SERIALIZE)` option should also be specified so that the service program can safely be called from a threaded Java application, for example:

```
H nomain thread(*serialize)
```

Figure 11-6 shows the prototyping Java method. This definition ensures that the Java and RPG binding signature match.

D dist	PR	4F	EXTPROC(*java: 'DistrictJ': 'dist')
D DISTID		10I 0	value

Figure 11-6 Prototyping Java method

As shown, dist is the name of RPG subprocedure, and it is linked to a Java method. The EXTPROC keyword indicates that the method name is dist(), and it is found in the DistrictJ class.

The data type is 4F telling the RPG compiler that the method returns a value of Java float.

```
D dist PR 4F EXTPROC(*java:
```

The data type of the parameter is specified as 10I. It is mapped to the Java int. Because the parameter is an int, it must be passed by value, so the VALUE keyword is required.

```
D DISTID 10I 0 value
```

Figure 11-7 shows the code of the native method.

P dist	B	EXPORT
D dist	PI	4F
D DISTID		10I 0 value
D YTDBAL	S	13P 2
C/EXEC SQL		
C+	select DYTD into :YTDBAL	
C+	from DSTRCT	
C+	where DID = :DISTID	
C/END-EXEC		
C	RETURN	YTDBAL
P dist	E	

Figure 11-7 The native method in RPG

The following line of code marks the beginning of the subprocedure. It shows that the subprocedure is to be exported into the service program. It is publicly available for access by the Java native method.

```
P dist B EXPORT
```

The prototype interface PI must match the prototype PR.

```
D dist PI 4F  
D DISTID 10I 0 value
```

We use an embedded SQL statement to retrieve the YTD balance from the DSTRCT table. The variable YTDBAL is defined to store the result of this SQL statement. It is returned to the caller of this procedure. You can also use native database I/O, rather than SQL, to read the file.

```

C/EXEC SQL
C+   select DYTD into :YTDBAL
C+       from DSTRCT
C+      where DID = :DISTID
C/END-EXEC

C           RETURN      YTDBAL

```

This completes the RPG program.

In this example, the RPG program receives an integer DISTID from a Java program and uses embedded SQL to retrieve the record from the DSTRCT file. It returns the RPG packed decimal field. RPG converts the packed decimal value to the required float return type so no extra conversion is required. The Java program writes the DISTID value and the RPG returned YTDBAL on the display of the caller of the Java program.

### 11.2.2 Calling a Java method from the RPG example

The Java Native Interface provides the ability to embed the function of a JVM into a native application. This facility is known as the *Invocation API*. This example demonstrates how to call Java methods from a RPG program. It also shows how to create Java objects in RPG. When the RPG program calls a Java method, RPG checks to see if the JVM has been started. If not, RPG starts the JVM and uses the value of the CLASSPATH environment variable for the classpath.

In this example, the following actions occur:

1. The RPG program, named CSTMRRINQR, creates the CstmrlnqJ Java object in its \*inzsr subroutine.
2. It accepts a customer number between one and ten from the display file.
3. A String object is created from the customer number. The String object is passed to the custName() method of the instance of the CstmrlnqJ class.
4. The customer number string is used to retrieve the row from the customer table (CSTMRR).
5. The result is returned as a String object, which is converted to a character field. The conversion is done by running the getBytes() method of the String object.
6. The final result is written to the display using the display file as shown in Figure 11-8.

```

Enter Customer No. 0001
The first name of Customer no 0001 is FERNANDO
F3=Exit

```

Figure 11-8 Running the CSTMRRINQR program to call Java methods

### The RPG program: CSTMRRINQR

The RPG program is shown in code snippets. We explain the statements regarding JNI and Java objects in the example.

Figure 11-9 shows the prototype of the constructor of the CstmrlnqJ class. Since the related method is a constructor, we must specify \*CONSTRUCTOR as the name of the method. This prototype is used to create the instance of CstmrlnqJ class. We specify the O in column 40 of the D-specification and use the CLASS keyword to provide the class of the object.

**Note:** A \*CONSTRUCTOR prototype does not require the CLASS keyword. The compiler knows that the returned object is the same as the specified class in the EXTPROC. It may be more convenient to leave it out.

```
d cstmrinq      pr          0  extproc(*JAVA:
d                           'CstmrlnqJ':
d                           *constructor)
d                           class(*java:'CstmrlnqJ')
```

Figure 11-9 Prototyping the constructor of the CstmrlnqJ class

We need to execute the custName() method in the CstmrlnqJ class. The prototype is shown in Figure 11-10. The parameter of the method is a String object, and the method returns a String object as well. Because the parameter is input-only, the CONST keyword is specified.

```
d custname      pr          0  extproc(*JAVA:
d                           'CstmrlnqJ':
d                           'custName')
d                           class(*java:'java.lang.String')
d   custno        0  class(*java:'java.lang.String')
d                           const
```

Figure 11-10 Prototyping the custName() method of the CstmrlnqJ class

**Note:** We use the STATIC keyword if that method is static.

We also need to create a String object in the RPG program. Figure 11-11 shows the prototype of the constructor of the String class. The constructor accepts a byte array as a parameter. Since Java byte[] cannot be declared with a fixed length, the VARYING keyword is specified. In our example, we arbitrarily specify the length of the parameter at 80 bytes.

```
d makestring    pr          0  extproc(*JAVA:
d                           'java.lang.String':
d                           *CONSTRUCTOR)
d   bytes        80A const varying
```

Figure 11-11 Prototyping the constructor of the String class

We convert a String object to a byte array using the getBytes() method of the String class. Then we can store the byte array in an alphanumeric field. This method has no parameter. The prototype is shown in Figure 11-12 on page 448.

```

d getBytes      pr          80A extproc(*JAVA:
d                               'java.lang.String':
d                               'getBytes')
d                               varying

```

Figure 11-12 Prototyping the `getBytes()` method of the `String` class

Figure 11-13 shows the RPG variables that are used to store the `CstmrInqJ` object and the `String` object respectively.

```

d cinq        s          0  class(*java:'CstmrInqJ')
d string      s          0  class(*java:'java.lang.String')

```

Figure 11-13 Defining the RPG variables to store the Java object

To call a non-static method, an object is required. Figure 11-14 shows how to create the `CstmrInqJ` object in the initialize subroutine of the RPG program. The `cinq` variable is used to store this object.

```

c     *inzsr      begsr
c           eval    cinq=cstmrinq()
c           endsr

```

Figure 11-14 The initialize subroutine of the `CSTMRINQR` program

Figure 11-15 shows the main routine of the `CSTMRINQR` program. The program gets the customer number (`CUSTNO`) from the display file. The `CUSTNO` is used to instantiate a `String` object. This `String` object is passed as a parameter to execute the `custName()` method. The method returns the result as a `String` object, which is converted by running the `getBytes()` method, to an alphanumeric field. The alphanumeric field (`RESULT`) is displayed on the screen.

```

c             exfmt   f01
c             dow     *in03=*off
c             eval    string=makestring(custno)
c             eval    string=custname(cinq:string)
c             eval    result=getBytes(string)
c             exfmt   f01
c             enddo
c             eval    *inlr=*on

```

Figure 11-15 The main routine of the `CSTMRINQR` program

If the method is not a static method, then it is called as an instance method. An object instance must be coded using an *extra* first parameter.

```

c             eval    string=custname(cinq:string)
c             eval    result=getBytes(string)

```

### The Java class: `CstmrInqJ`

The Java class, shown in Figure 11-16, is called by the RPG program. The class uses the JDBC statements to retrieve the row from the database.

When the class constructor is called, the JDBC driver is loaded. The dbConnect Connection object is created by running the getConnection() static method of the DriverManager class. The psFirstName PreparedStatement object is then created by executing the prepareStatement() method of the dbConnect object. This process is to prepare the SQL statement to retrieve the CFIRST column from the CSTMTR table. If the process is successful, we are ready to access the database.

The custName() method accepts a customerNo String object. The customerNo is used to set the first parameter of the psFirstName PreparedStatement object. The executeQuery() method is executed to select the row from the CSTMTR table. The value of the CFIRST column is retrieved and returned as a String object.

```

import java.sql.*;

public class CstmrInqJ
{
    static public String USER          = "*current";
    static public String PASSWORD      = "*current";
    static public String SYSTEM        = "localhost";
    static public String LIBRARY       = "apilib";
    private PreparedStatement psFirstName;

    public CstmrInqJ()
    {
        try
        {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            Connection dbConnect =
                DriverManager.getConnection("jdbc:db2://"+SYSTEM+"/"+LIBRARY,
                                           USER, PASSWORD);
            psFirstName = dbConnect.prepareStatement(
                "select cfirst from cstmr where cid=?");
        }
        catch(Exception e)
        {
            e.printStackTrace();
            System.exit(-1);
        }
    }

    public String custName (String customerNo) throws SQLException
    {
        String customerName = "not found";
        psFirstName.setString(1, customerNo);
        ResultSet rs = psFirstName.executeQuery();
        if (rs.next()) customerName = rs.getString("cfirst");
        return "The first name of Customer no "+customerNo+" is "+customerName;
    }
}

```

*Figure 11-16 The CstmrInq Java class*

In summary, the RPG program accepts a customer number as input and passes it as a Java String object to the Java program. The Java program uses this field to retrieve a record from the customer table using the JDBC statements. The String result is consolidated and returned to the RPG program. This resulting Java String is then converted to a alphanumeric field and written to the display by the RPG program.

### 11.2.3 RPG and JNI consideration

In V5R1, the RPG compiler makes the interfacing between RPG and Java easy and hides almost all the JNI coding from the programmer. However, we may need to do the actual JNI coding for the following reasons:

- ▶ Improved performance. For example, RPG always converts arrays between RPG and Java on calls and on entry and exit from native methods. We may want to handle our own array conversions.
- ▶ We may want to directly access fields in a class object.

The JNI header file is provided as a source member named JNI in QSYSINC/QRPGLESRC. We use the /COPY statement to copy this file into the RPG program. For detailed information, refer to “Advanced JNI coding” in *ILE RPG for AS/400 V5R1 Programmer’s Guide*, SC09-2507.

## 11.3 RPG Order Entry example

In this example, the iSeries Order Entry application is changed to use RPG and embedded SQL. JNI is used to call RPG sub-procedures from the Java program. These programs are based on the programs discussed in 9.7, “Order Entry using JDBC” on page 388. The Java server program that runs on the iSeries server is invoked using remote method invocation (RMI) from a Java client program. The RMI invoked iSeries server Java program uses JNI to invoke RPG sub-procedures. Figure 11-17 shows the new design.

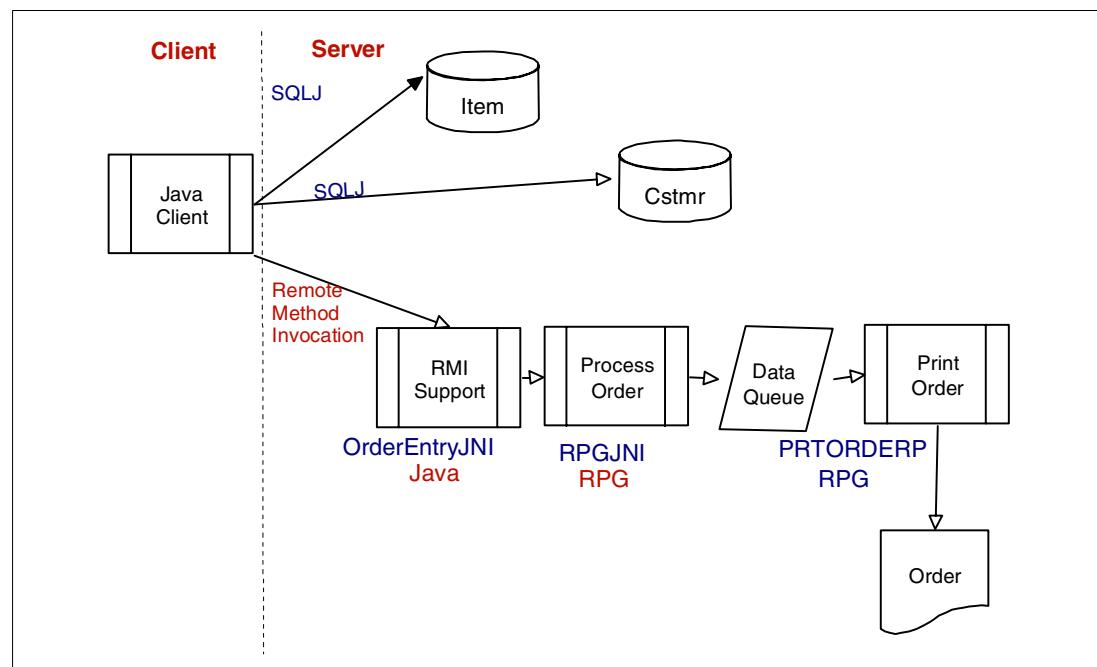


Figure 11-17 Order Entry RPG JNI example

The Java server application is based on the JDBC application named OrderEntryJDBC found in the serverOrderEntry package. The new application name is OrderEntryJNI. The class and its methods are shown in Figure 11-18.

```

serverOrderEntry Package
  OrderEntryJNI Class
    commitOrder()
    addOrderLine()
    addOrderHeader()
    updateStock()
    updateCustomer()
    writeDataQueue()
    getOrderNumber()
    finalize()
    initialize()
    writeOrderLine()

```

*Figure 11-18 The OrderEntryJNI class*

### 11.3.1 Converting the Order Entry application to use JNI

Figure 11-19 shows the class definition for the JNI version. We only keep the class variables necessary for the program.

```

import java.math.*;
import java.rmi.*;
import java.rmi.server.*;
import com.ibm.as400.access.*;
import orderObjects.*;

public class OrderEntryJNI extends UnicastRemoteObject implements OrderEntryI
{
    private static final String SYSTEM = "localhost";
    private static final String USER= "user id";
    private static final String PASSWORD= "password";
    private static final String SYSTEM_LIBRARY = "QSYS.LIB";
    private static final String DATA_QUEUE_NAME= "ORDERS.DTAQ";
    private static final String DATA_QUEUE_LIBRARY= "APILIB.LIB";
    private static final String WAREHOUSE= "0001";
    private static final int DISTRICT= 1;
    private AS400 as400= null;
}

```

*Figure 11-19 The class definition of OrderEntryJNI*

### Loading the RPG service program

Figure 11-20 on page 452 shows how the RPGJNI service program is loaded by the Java program. The code is in the class declaration.

```

static
{
    try
    {
        System.out.println("Loading RPG service program RPGJNI");
        System.loadLibrary("RPGJNI");
        System.out.println("Loaded RPG service program RPGJNI");
    }
    catch(Exception e)
    {
        System.out.println("Could not load RPG program RPGJNI");
    }
}

```

*Figure 11-20 Loading the RPGJNI service program*

### The initialize() method

The as400 object is still used when we instantiate IBM Toolbox for Java objects in the writeDataQueue() method. We remove the statements used by the JDBC version.

```

public void initialize()
{
    as400 = new AS400(SYSTEM, USER, PASSWORD);
}

```

*Figure 11-21 The initialize() method*

### The commitOrder() method

We only remove the JDBC commit statements from this method.

```

// Commit the database changes
if (dbConnection.getTransactionIsolation() != java.sql.Connection.TRANSACTION_NONE)
{
    dbConnection.commit();
}

```

### The addOrderHeader() method

We change this method to call the RPG native method.

```

native void addOrderHeader(String aCustomerNumber,
                           BigDecimal anOrderNumber,
                           BigDecimal anOrderLineCount);

```

*Figure 11-22 The addOrderHeader method*

### The addOrderLine() method

We add a new native method, writeOrderLine(), to insert rows to the ORDLIN table on the iSeries server. The writeOrderLine() method is executed from the addOrderLine() method. It provides ten parameters. The parameters are:

- ▶ Order number
- ▶ District
- ▶ Warehouse
- ▶ Order line number

- ▶ String constant "JAVA"
- ▶ Item id
- ▶ Item quantity
- ▶ Order amount
- ▶ Date
- ▶ Time

```

private BigDecimal addOrderLine(BigDecimal anOrderNumber, Order anOrder) throws
Exception
{
    BigDecimal orderTotal = new BigDecimal(0);
    int linecount = 0;

    BigDecimal customerDiscount = getCustomerDiscount(anOrder.getCustomerId());

    java.util.Enumeration e = anOrder.getOrderDetail().elements();
    // While we have order lines to process
    while(e.hasMoreElements())
    {
        OrderDetail orderLine = (OrderDetail) e.nextElement();

        BigDecimal orderAmount = (orderLine.getItemPrice().subtract(
            orderLine.getItemPrice().multiply(customerDiscount).
            divide(new BigDecimal(100), BigDecimal.ROUND_DOWN))).
            multiply(orderLine.getItemQty()).setScale(2, BigDecimal.ROUND_HALF_UP);

        float adate =12311999;
        float atime =235959;

        writeOrderLine(anOrderNumber, DISTRICT, WAREHOUSE, ++linecount, "JAVA",
                      orderLine.getItemId(), orderLine.getItemQty(), orderAmount,
                      adate, atime);

        orderTotal = orderTotal.add(orderAmount);

        updateStock(orderLine.getItemId(), orderLine.getItemQty());
    }
    return orderTotal;
}

```

*Figure 11-23 The addOrderLine() method*

### The getCustomerDiscount() method

We change this method to call the RPG native method.

```

native BigDecimal getCustomerDiscount (String aCustomerId);

```

*Figure 11-24 The getCustomerDiscount() method*

### The getOrderNumber() method

We change this method to call the RPG native method.

```

native private BigDecimal getOrderNumber();

```

*Figure 11-25 The getOrderNumber() method*

## The updateCustomer() method

We change this method to call the RPG native method.

```
native void updateCustomer(String aCustomerID, BigDecimal anOrderTotal);
```

Figure 11-26 The updateCustomer() method

## The updateStock() method

We change this method to call the RPG native method.

```
native void updateStock(String aPartNbr, BigDecimal aPartQty);
```

Figure 11-27 The updateStock() method

## The writeDataQueue() method

No changes are required for this method.

## The getRawDate() method

This method has been removed.

## The getRawTime() method

This method has been removed.

## The finalize() method

The JDBC close() method statements are removed. We only use the as400 object.

```
protected void finalize( ) throws Throwable
{
    // Close the AS400 object
    if (as400.isConnected())
    {
        as400.disconnectAllServices();
        as400 = null;
    }
    super.finalize();
    return;
}
```

Figure 11-28 The finalize() method

## The writeOrderLine() method

This is a native method used to insert a row into the ORDLIN table.

```
native void writeOrderLine(BigDecimal ordernbr, int district, String warehouse,
                           int linecount, String ajava, String itemid,
                           BigDecimal itemqty, BigDecimal orderamt,
                           float adate, float atime);
```

Figure 11-29 The writeOrderLine() method

### 11.3.2 The RPG service program

The RPG service program is called RPGJNI. It contains sub-procedures that are invoked from the Java server program. These RPG sub-procedures are invoked:

- ▶ OrdNbr
- ▶ Discount
- ▶ OrdLin
- ▶ UpdStock
- ▶ addOrdHdr
- ▶ UpdCst

#### Prototyping Java methods in RPG

In the following code snippets, the Java native method definition and the RPG prototype definition are shown together. The RPG compiler must know the name of the method, the class to which it belongs, the data types of the parameters, and the data type of the returned value (if any).

##### ▶ The getOrderNumber() method

Figure 11-30 shows the Java native method declaration of `getOrderNumber()` and the matching RPG sub-procedure prototype `OrdNbr`.

```
native private BigDecimal getOrderNumber();  
  
D OrdNbr      PR          0  EXTPROC(*java:  
D                      'serverOrderEntry.OrderEntryJNI':  
D                      'getOrderNumber')  
D                      class(*java:'java.math.BigDecimal')
```

Figure 11-30 The `getOrderNumber()` method and its prototype

##### ▶ The getCustomerDiscount() method

Figure 11-31 shows the Java native method declaration of `getCustomerDiscount()` and the matching RPG sub-procedure prototype `Discount`.

```
native BigDecimal getCustomerDiscount (String aCustomerId);  
  
D Discount      PR          0  EXTPROC(*java:  
D                      'serverOrderEntry.OrderEntryJNI':  
D                      'getCustomerDiscount')  
D                      class(*java:'java.math.BigDecimal')  
D  customerId      0  class(*java:'java.lang.String')
```

Figure 11-31 The `getCustomerDiscount()` method and its prototype

##### ▶ The updateStock() method

Figure 11-32 on page 456 shows the Java native method declaration of `updateStock()` and the matching RPG sub-procedure prototype `UpdStock`.

```

native void updateStock(String aPartNbr, BigDecimal aPartQty);

D UpdStock      PR          EXTPROC(*java:
D                      'serverOrderEntry.OrderEntryJNI':
D                      'updateStock')
D aPartNbr       0           class(*java:'java.lang.String')
D aPartQty       0           class(*java:'java.math.BigDecimal')

```

*Figure 11-32 The updateStock() method and its prototype*

► **The addOrderHeader() method**

Figure 11-33 shows the Java native method declaration of addOrderHeader() and the matching RPG sub-procedure prototype addOrdHdr.

```

native void addOrderHeader(String aCustomerNumber,
                           BigDecimal anOrderNumber,
                           BigDecimal anOrderLineCount);

D addOrdHdr      PR          EXTPROC(*java:
D                      'serverOrderEntry.OrderEntryJNI':
D                      'addOrderHeader')
D cstNbr         0           class(*java:'java.lang.String')
D ordNbr         0           class(*java:'java.math.BigDecimal')
D linecount       0           class(*java:'java.math.BigDecimal')

```

*Figure 11-33 The addOrderHeader() method and its prototype*

► **The updateCustomer() method**

Figure 11-34 shows the Java native method declaration of updateCustomer() and the matching RPG sub-procedure prototype updCst.

```

native void updateCustomer(String aCustomerID, BigDecimal anOrderTotal);

D Updcst        PR          EXTPROC(*java:
D                      'serverOrderEntry.OrderEntryJNI':
D                      'updateCustomer')
D cstNbr         0           class(*java:'java.lang.String')
D oTotal         0           class(*java:'java.math.BigDecimal')

```

*Figure 11-34 The updateCustomer() method and its prototype*

► **The writeOrderLine() method**

Figure 11-35 shows the Java native method declaration of writeOrderLine() and the matching RPG sub-procedure prototype OrdLin.

```

native void writeOrderLine(BigDecimal ordernbr, int district, String warehouse,
                           int linecount, String ajava, String itemid,
                           BigDecimal itemqty, BigDecimal orderamt,
                           float adate, float atime);

D Ordlin      PR          EXTPROC(*java:
D                      'serverOrderEntry.OrderEntryJNI':
D                      'writeOrderLine')
D ordernbr      0  class(*java:'java.math.BigDecimal')
D district      10I 0 value
D warehouse     0  class(*java:'java.lang.String')
D linecnt      10I 0 value
D ajava        0  class(*java:'java.lang.String')
D itemid       0  class(*java:'java.lang.String')
D itemqty      0  class(*java:'java.math.BigDecimal')
D orderamt     0  class(*java:'java.math.BigDecimal')
D adate        4F   value
D atime        4F   value

```

Figure 11-35 The `writeOrderLine()` method and its prototype

### **Additional prototypes for Java methods**

In the RPG service program, we need to execute the Java methods. Some of them were explained in earlier.

```

d makestring    pr          0  extproc(*JAVA:
d                      'java.lang.String':
d                      '*CONSTRUCTOR')
d bytes         80A const varying

```

Figure 11-36 Prototyping the constructor of the `String` class

```

d getBytes      pr          80A extproc(*JAVA:
d                      'java.lang.String':
d                      'getBytes')
d                  varying

```

Figure 11-37 Prototyping the `getBytes()` method of the `String` class

Figure 11-38 shows the prototype of the constructor of the `BigDecimal` class. It is used to create a `BigDecimal` object in the RPG program. A value of `double` is passed as a parameter.

```

d makebigd     pr          0  extproc(*JAVA:
d                      'java.math.BigDecimal':
d                      '*CONSTRUCTOR')
d double       8F   value

```

Figure 11-38 Prototyping the constructor of the `BigDecimal` class

The `floatValue()` method is called when we need to convert a Java `BigDecimal` object to a float value. Figure 11-39 on page 458 shows the prototype of this method.

```

d floatValue      pr          4F  extproc(*JAVA:
d                           'java.math.BigDecimal':
d                           'floatValue')

```

*Figure 11-39 Prototyping the floatValue() method of the BigDecimal class*

## The RPG subprocedures

In the following code snippets, the complete RPG code of the subprocedures are shown. The details of the RPG statements were already explained in the earlier section.

### ► The OrdNbr subprocedure

Figure 11-40 shows the complete code for the OrdNbr subprocedure. It has no parameters. The returned value, the order number, is a Java BigDecimal object that is created using the double value DNXTORA.

```

P OrdNbr      B          EXPORT
D             PI          0  class(*java:'java.math.BigDecimal')

DDNXTORA      S          8F
DDNXTORB      S          8F

C/EXEC SQL
C+   SELECT DNXTOR INTO :DNXTORA FROM DSTRCT
C+   WHERE DID = 1 AND DWID = '0001'
C/END-EXEC
C           EVAL      DNXTORB = DNXTORA + 1
C/EXEC SQL
C+   UPDATE DSTRCT SET DNXTOR = :DNXTORB
C+   WHERE DID = 1 AND DWID = '0001'
C/END-EXEC
C           return    makebigd(DNXTORA)
P OrdNbr      E

```

*Figure 11-40 The OrdNbr subprocedure*

► **The Discount subprocedure**

Figure 11-41 shows the complete code for the Discount procedure. A Java String, customerId, is passed as the parameter. The returned value, the discount, is a Java BigDecimal object that is created using the double value cdiscount.

```
P Discount      B          EXPORT
D             PI          0  class(*java:'java.math.BigDecimal')
D  customerId           0  class(*java:'java.lang.String')

D CustEBCDIC    S          4A
D cdiscount     S          8F

C                  EVAL      CustEBCDIC = getBytes(customerId)
C/EXEC SQL
C+  select CDCT into :cdiscoun
C+      from CSTMR
C+      where CID = :CustEBCDIC
C/end-exec
C          return      makebigd(cdiscount)
P Discount      E
```

Figure 11-41 The Discount subprocedure

► The OrdLin subprocedure

Figure 11-42 shows the complete code for the OrdLin subprocedure. This subprocedure inserts a row to the ORDLIN table using the passed parameters. The parameters used are:

- The order number is a Java BigDecimal object.
- The district is an integer value.
- The warehouse is a Java String object.
- The line count is an integer value.
- The constant “JAVA” is a Java String object.
- The item id is a Java String object.
- The item quantity is a Java BigDecimal object.
- The order amount is a Java BigDecimal object.
- The date is a float value.
- The time is a float value.

There is no returned value from this procedure.

```

P Ordlin      B          EXPORT
D             PI
D orderNbr           0   class(*java:'java.math.BigDecimal')
D district          10I 0 value
D warehouse         0   class(*java:'java.lang.String')
D linecnt          10I 0 value
D ajava             0   class(*java:'java.lang.String')
D itemid            0   class(*java:'java.lang.String')
D itemqty           0   class(*java:'java.math.BigDecimal')
D orderamt          0   class(*java:'java.math.BigDecimal')
D adate             4F   value
D atime             4F   value

D warehouseE        S       4A
D ajavaE            S       4A
D itemidE           S       6A
D ordno              S       4F
D itmqty             S       4F
D ordamt             S       4F

C                   EVAL    warehouseE = getBytes(warehouse)
C                   EVAL    ajavaE = getBytes(ajava)
C                   EVAL    itemidE = getBytes(itemid)
C                   EVAL    ordno = floatValue(orderNbr)
C                   EVAL    itmqty = floatValue(itemqty)
C                   EVAL    ordamt = floatValue(orderamt)

C/exec sql
C+ insert into ordlin
C+ (OLOID, OLDID, OLWID, OLNBR, OLSPWH,
C+ OLIID, OLQTY, OLAMNT, OLDVD, OLDLVT)
C+ values(
C+ :ordno, :district, :warehouseE, :linecnt, :ajavaE,
C+ :itemidE, :itmqty, :ordamt, :adate, :atime )
C/end-exec
C               return
P Ordlin      E

```

Figure 11-42 The OrdLin subprocedure

► **The UpdStock subprocedure**

Figure 11-43 shows the complete code for the UpdStock subprocedure. This subprocedure accepts a Java String, the part number, and a Java BigDecimal, the part quantity, as the parameters. The STOCK table is updated using the passed parameters.

```
P UpdStock      B          EXPORT
D             PI
D  aPartNbr      0  class(*java:'java.lang.String')
D  aPartQty      0  class(*java:'java.math.BigDecimal')

D aPartNbrE      S          6A
D qty            S          4F
D tqty           S          4F

C               EVAL      aPartNbrE = getBytes(aPartNbr)
C               EVAL      qty      = floatValue(aPartQty)
C/Exec SQL
C+   select STQTY into :tqty
C+     from stock
C+   where STIID = :aPartNbrE
C+   and   STWID = '0001'
C/end-exec
C               EVAL      tqty = tqty - qty
C/Exec SQL
C+   update STOCK
C+     set      STQTY = :tqty
C+     where STIID    = :aPartNbrE
C+     and    STWID    = '0001'
C/end-exec
C               return
P UpdStock      E
```

Figure 11-43 The UpdStock subprocedure

► **The addOrdHdr subprocedure**

Figure 11-44 shows the complete code for the addOrdHdr subprocedure. This subprocedure inserts a row into the ORDERS table using passed parameters. The parameters used are:

- The customer number is a Java String object.
- The order number is a Java BigDecimal object.
- The line count is a Java BigDecimal object.

```
P addOrdHdr      B          EXPORT
D                  PI
D cstNbr          0  class(*java:'java.lang.String')
D ordNbr          0  class(*java:'java.math.BigDecimal')
D lineCount        0  class(*java:'java.math.BigDecimal')

D cstNbrE         S          4A
D ordno           S          4F
D lincnt          S          4F
D DS
Dtimedate        14  0
Dtime             S          6  0
Ddate             S          8  0

C                 EVAL      cstNbrE = getBytes(cstNbr)
C                 EVAL      ordno   = floatValue(ordNbr)
C                 EVAL      lincnt = floatValue(lineCount)
C                 time      timedate
C                 move1    timedate   time
C                 move     timedate   date
C/exec SQL
C+   insert into ORDERS
C+   (OWID, ODID, OCID, OID, O_LINES,
C+   OCARID, O_LOCAL, OENTDT, OENTTM)
C+   values
C+   ('0001', 1, :cstNbrE, :ordno, :lincnt,
C+   'ZZ', 1, :date, :time)
C/end-exec
C                 RETURN
P addOrdHdr      E
```

Figure 11-44 The addOrdHdr subprocedure

► **The UpdCst subprocedure**

Figure 11-45 shows the complete code for the UpdCst subprocedure. This subprocedure updates the customer row in the CSTMR table. The Java String, the customer number, and the Java BigDecimal, the order total, are passed as the parameters.

```
P UpdCst      B          EXPORT
D             PI
D cstNbr           o  class(*java:'java.lang.String')
D orderTotal        o  class(*java:'java.math.BigDecimal')

D             DS
D timedate        14  0
D time            S       6  0
D date            S       8  0
D cstNbrE         S       4A
D oTotal          S       4F
D bal             S       4F
D ytd             S       4F

C             EVAL    cstNbrE = getBytes(cstNbr)
C             EVAL    oTotal   = floatValue(orderTotal)

C/exec sql
C+   select CBAL, CYTD
C+   into  :bal, :ytd
C+   from CSTMR
C+   where CID = :cstNbrE
C/end-exec
C             time          timedate
C             move1   timedate     time
C             move    timedate     date
C             EVAL    bal = bal + oTotal
C             EVAL    ytd = ytd + oTotal

C/exec sql
C+   update CSTMR
C+   set CLDATE = :date, CLTIME = :time,
C+         CBAL   = :bal, CYTD   = :ytd
C+   where CID = :cstNbrE
C/end-exec
C             RETURN
P UpdCst      E
```

Figure 11-45 The UpdCst subprocedure

## 11.4 Java Native Interface and C

This section discusses using the Java Native Interface and C on the iSeries server.

**Parameter passing:** There are problems with parameters passed by value of types jfloat, jshort, jchar, and jbyte that are caused by parameter widening that is done by Java and C, but not by RPG. We recommend that you *do not use* float and short. Instead, replace them with double and int.

Dealing with char and byte is trickier. It is best to avoid them, or pass them from or to Java as an unsigned int. The problems do not always show up since the translator passes primitives in registers whenever possible. Using a register for the parameter hides the problem. Unfortunately, the problem shows up when the call statement becomes complex enough to use up all of the registers. Therefore, some of the primitive parameters have to be picked up from the stack.

### 11.4.1 Setting up JNI and C

No special prerequisite source members are required to use JNI and C. The C and JNI for C header files, which are actually source members, are included as part of OS/400 and the C Licensed Program Product. These source members are copied into a C program when using JNI. The source files are available on the iSeries server in the files shown in Table 11-1 and Table 11-2.

Table 11-1 Prior to V5R1 source members and their locations on the iSeries server

Source member	Location on the iSeries server
jni.h	QSYSINC/H/JNI
QJAVA/H/JNI	QJAVA/H/JNI
stdio.h	QCLE/H/STDIO
stdarg.h	QCLE/H/STDARG
jni_md.h	QSYSINC/H/JNI_MD
QJAVA/H/JNI_MD	QJAVA/H/JNI_MD

In V5R1, the source files are in H file in the QSYSINC library.

Table 11-2 V5R1 source members and their locations on the iSeries server

Source member	Location on the iSeries server
jni.h	QSYSINC/H/JNI
stdio.h	QSYSINC/H/STDIO
stdarg.h	QSYSINC/H/STDARG
jni_md.h	QSYSINC/H/JNI_MD

Once the source members are located, follow these steps to use JNI and C:

1. Create a source physical file in your own library to store your own Java program header file members. For any user-written Java program that is to be linked with C, you must create a header file and store it in the source physical file. The command to create the header source physical file is shown in Figure 11-46.

*Figure 11-46 Create Source Physical File (CRTSRCF)*

2. Generate a header file for each Java class that is to use JNI. Make sure that a class file exists before you create your header file. For example, to create a JNI header file for the HelloJ program, enter this command:

javah -jni HelloJ

The Java class files and header files are stored in an iSeries integrated file system directory.

3. Convert the header file from the integrated file system into an iSeries library as a member of the source physical file H. The previous step shows how to create the H file in your own library. Figure 11-47 shows the command to copy and convert from the integrated file system to a library file member.

```
Copy From Stream File (CPYFRMSTMF)

Type choices, press Enter.

From stream file . . . . . > '/home/a999501a>HelloJ.h'

To file member or save file . . > '/qsys.lib/apilib.lib/h.file/HelloJ.mbr'

Member option . . . . . > *ADD           *NONE, *ADD, *REPLACE
Data conversion options . . . *AUTO        *AUTO, *TBL, *NONE
Stream file code page . . . *STMF        1-32767, *STMF, *PCASCII
Database file CCSID . . . *FILE        1-65533, *FILE
End of line characters . . . *ALL         *ALL, *CRLF, *LF, *CR...
Tab character expansion . . . *YES        *YES, *NO
```

*Figure 11-47 Copy From Stream File (CPYFRMSTMF)*

Figure 11-48 on page 466 shows an example of how the header file looks for the HelloJ Java program.

```

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloJ */

#ifndef _Included_HelloJ
#define _Included_HelloJ
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      HelloJ
 * Method:     helloCmethod
 * Signature:  ()V
 */
JNIEXPORT void JNICALL Java_HelloJ_helloCmethod
    (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif

```

*Figure 11-48 Header file for the HelloJ Java program*

The method name is `Java_HelloJ_helloCmethod`. *HelloJ* is the name of the Java program, and *helloCmethod* is the name of the native method in Java.

Also, note the parameter list. A minimum of two parameters are required, a `JNIEnv` pointer and a `jobject` object, as shown in `(JNIEnv *, jobject)`.

Any additional parameters that are passed between the Java program and the C program also appear in the parameter list. The header file is included in the C program, and the above signature is used.

## 11.4.2 Hello C example

In this example, a Java program, named *HelloJ*, creates a new instance of a C native method, called *helloCmethod*. The C program starts when this native method is instantiated by the Java program. The C program prints a message on the display. This simple example demonstrates Java and C method calls, so you can learn the basics of using JNI and C.

### Java class: HelloJ

This program is the same as the one used to start the RPG procedure, which is explained in detail in the Java portion of 11.3.2, “The RPG service program” on page 455. On the Java side, it makes no difference at all whether an RPG or C service program is being invoked or what language the native method uses. In Figure 11-49, the service program is called HELLOC, which is based on a C module. This service program can also include RPG modules, which are supported by the iSeries ILE. The native method being invoked is `helloCmethod()`.

```

public class HelloJ
{
    public native void helloCmethod();

    static
    {
        System.loadLibrary("HELLOC");
    }

    public static void main(String args[])
    {
        new HelloJ().helloCmethod();
    }
}

```

*Figure 11-49 HelloJ Java class*

### **Header file for Java program: HelloJ**

The header file is explained in 11.4.1, “Setting up JNI and C” on page 464, which shows how to make your header file available on the iSeries server.

### **C program: HelloC**

An example of a C program, named HelloC, is shown in Figure 11-50.

```

#include "HelloJ.h"

JNIEXPORT void JNICALL Java_HelloJ_helloCmethod
(JNIEnv *env, jobject javaThis)
{
    printf("Hello from AS/400 C. Have a good day.\n");
    return;
}

```

*Figure 11-50 HelloC C program*

The `#include "HelloJ.h"` statement establishes the link between the C program and Java class file through the Java header file. The header file, named HelloJ, is listed and explained in 11.4.1, “Setting up JNI and C” on page 464. As explained in the setup instructions, this header file must be copied from the integrated file system to a source file member in the H file in your library.

It is also important to match of the signature of the header file and the C program, as shown here:

```

JNIEXPORT void JNICALL Java_HelloJ_helloCmethod
JNIEnv *env, jobject javaThis)

```

The arguments in the signature can vary, because they depend on the number and type of parameters that are being used. Figure 11-50 shows the minimum number of signature parameters. There must be at least two parameters: the JNI environment pointer and the Java class object.

### 11.4.3 Changing a Java String object from a C program

In this Java program, a String **s** is defined, but is not initialized. Using JNI functions, this Java object is made available to a C program. The C program receives a pointer to the Java String object **s** and sets the Java String to another value from within the C program.

This program demonstrates the capability of JNI to make Java objects available to C and to allow C to manipulate Java objects. The same thing can also be done using JNI and RPG.

#### Java class: ChangeStgJ

Figure 11-51 shows the Java program that loads the iSeries service program, named ChangeStgC, and calls the C native method named setTheString. The native method or the service program can be RPG or C++. The Java program remains the same because it makes no difference to Java what the language of the native method is.

```
public class ChangeStgJ
{
    public String s;
    public native void setTheString();
    static
    {
        System.loadLibrary("ChangeStgC");
    }

    public static void main(String argv[])
    {

        ChangeStgJ cs = new ChangeStgJ();

        System.out.println("String field is '" + cs.s + "'");

        cs.setTheString();

        System.out.println("String field is '" + cs.s + "'");
    }
}
```

Figure 11-51 ChangeStgJ Java program

In this statement, a String **s** is defined:

```
public String s;
```

A String **s** is defined. However, without giving it a value, it is printed in this statement:

```
System.out.println("String field is '" + cs.s + "'")
```

The above statement displays null as the String value, because String **s** has not been given a value.

This statement runs the native method, called setTheString, which is written in the C language to set a value for the String:

```
cs.setTheString();
```

No parameters are passed to the native method. So, how can we expect to set a parameter in an invoked method without passing it as an argument?

This example shows the power of JNI. The C program actually accesses the Java String object storage location using JNI and changes its value.

The next execution of the `System.out.println("String field is " + cs.s + "")` statement prints the value of the String as set by the C program.

### Header file for the Java program: ChangeStgC

Figure 11-52 shows the header file for the ChangeStgC Java program. You must create this file for the Java class. Then, copy the source member of the H file into your library for the C program to interface with the Java program. For more details, see 11.4.1, “Setting up JNI and C” on page 464.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class ChangeStgJ */

#ifndef _Included_ChangeStgJ
#define _Included_ChangeStgJ
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     ChangeStgJ
 * Method:    setTheString
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_ChangeStgJ_setTheString
    (JNIEnv *, jobject);
#endif
/*
 * Class:     ChangeStgJ
 * Method:    setTheString
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_ChangeStgJ_setTheString
    (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Figure 11-52 Header file for the *ChangeStgJ* Java program

### C program: ChangeStgC

The complete C program is shown in segments. Only new concepts and statements that are introduced in this example are explained.

Figure 11-53 on page 470 shows the beginning code of the C program. The details for each line of code are explained in the text that follows the figure.

```

#include "ChangeStgJ.h"

#pragma convert(819)

JNIEXPORT void JNICALL Java_ChangeStgJ_setTheString
(JNIEnv *env, jobject javaThis)
{
    jclass      thisClass;
    jstring     stringObject;
    jfieldID   fid;

    stringObject = (*env) -> NewStringUTF(env,
        "Hello, this Java String Object is changed by C using JNI");
}

```

*Figure 11-53 ChangeStgC C program*

The `#pragma convert(819)` statement changes the language code page that is used in the program to ASCII. This is done, so String literals are treated as ASCII and are easily converted into Java String objects that use JNI functions. If you do not do this, the program has to convert the literal from EBCDIC to ASCII before using it with JNI, because EBCDIC is the default language of the iSeries server. If you want to use EBCDIC in the program, use `convert(0)` with the `#pragma` statement. The default is needed if you want to use statements, such as `printf`, to print messages on the iSeries display.

The code segment in Figure 11-53 defines these variables:

```

jclass      thisClass;
jstring     stringObject;
jfieldID   fid;

```

These variable types (`jclass`, `jstring`, and so on) are available because the first statement of this C program includes the `ChangeStgJ.h` header file. The first statement after the comments in this header file is:

```
#include <jni.h>
```

The `jni.h` file is copied into the C program. It contains definitions for all of the JNI variable types (`jclass`, `jstring`, and so on).

This is a JNI function:

```

stringObject = (*env) -> NewStringUTF(env,
    "Hello, this Java String is changed by C using JNI");

```

The `NewStringUTF` function converts an ASCII string into a Java String object. It uses an address pointer to store the object in the Java environment as a String object. The variable `stringObject` was defined as type `jstring` earlier. It is now initialized with a String value.

Figure 11-54 shows the JNI function code for this program.

```

thisClass = (*env) -> GetObjectClass(env, javaThis);

fid = (*env) -> GetFieldID(env, thisClass,
                           "s", "Ljava/lang/String;");

(*env) -> SetObjectField(env, javaThis, fid, stringObject);
}

```

Figure 11-54 The JNI function code

The JNI function, called `GetObjectClass`, which uses the JNI environment pointer, allows us to store the Java program object, named `ChangeStgJ`, as a variable `thisClass` in the C program. This statement also makes the Java object available inside of the C program. You can also use this function in RPG or C++ to make Java program objects available in these languages.

The JNI function, called `GetFieldID`, makes the String `s`, as defined in `thisClass`, available as a field `fid` of type `jstring`.

**Note:** You must specify the signature of a String object "`Ljava/lang/String;`" to make this retrieval possible.

The JNI function, called `SetObjectField`, actually changes the String `s`, based on the information that was retrieved in the previous JNI function. The information required for this function is:

- ▶ **stringObject:** Actual message we want to set
- ▶ **fid:** Location of the String `s`
- ▶ **javaThis:** Contains the `ChangeStgJ` Java program object
- ▶ **env:** JVM environment where these items are stored

In summary, this C program was invoked from Java, but no parameter was passed to it. The C program used JNI function calls to access the Java class object, retrieve the String `s` from the Java class, set the String to an ASCII value, and finally change the String `s` in the Java program with this value. After the C program is done processing, the Java program prints the String `s` on the display. It displays the value that was set by the C program.

This example demonstrates how you can make Java classes available to C. It also explains how Java objects can be retrieved and manipulated in C. Of course, both RPG and C++ can also do this using JNI functions.

#### 11.4.4 C with the Java Invocation API

In this example, we demonstrate invoking a Java program from C. This example is completely different than the previous examples. The following actions *are not* performed:

- ▶ Writing a Java program.
- ▶ Creating a header file.
- ▶ Creating a C or RPG module and then making a service program out of it. We simply write a C program.
- ▶ Using a service program. Instead, we use a normal executable C program.
- ▶ Starting a Java program. Instead, we start a C executable program, so no JVM is available.

In this example, we demonstrate:

- ▶ Using a C program to create a JVM using a JNI function.
- ▶ Locating where the Java programs are stored in the integrated file system.
- ▶ Specifying the Java program to be loaded and the method within that Java program to be called.
- ▶ Calling a method of a Java program.

For this scenario, we load the ChangeStgJ Java program into the JVM and start the main method of this Java program.

Any job that creates a JVM must be multi-thread capable. The only jobs on the iSeries server that are multi-thread capable are batch immediate (BCI) jobs. This program needs to be submitted in batch for execution by the Submit Job (SBMJOB) command. The command is:

```
SBMJOB CMD(CALL PGM(APILIB(INVOKEJAVA)) ALWMLTTHD(*YES)
```

The code is very powerful, yet generic in nature. If we change the name of the class, a different Java program is loaded. Similarly, if we change the name of the method, a different Java method is executed. This type of programming is called *Java Invocation API*.

#### 11.4.5 C program: INVOKEJAVA

The complete INVOKEJAVA C program example is explained here by showing code snippets.

In the earlier C examples, the user created header file was included by a `#include` statement. This copied the JNI header file into the program. In this example, there is no user created header file used. We have to explicitly include the header files shown in Figure 11-55.

```
#include <stdlib.h>
#include <string.h>
#include <jni.h>

int main (int argc, char *argv??(??))
{
    JDK1_1InitArgs initArgs;
    JavaVM*        myJVM;
    JNIEnv*        myEnv;
    char*          myClasspath;
    jclass         myClass;
    jmethodID      mainID;
    jclass         stringClass;
    jobjectArray   args;

    initArgs.version = 0x00010001;
```

Figure 11-55 INVOKEJAVA C program

The declaration of the main procedure should look like this:

```
int main (int argc, char *argv[])
```

Brackets ([ ]) are not available with the native iSeries editor. Instead of using these brackets, we use an equivalent symbol. The equivalent symbols are:

- ▶ [ is equivalent to ??(
- ▶ ] is equivalent to ??)

Various variables are defined. For example, JavaVM denotes a Java virtual machine, and JNIEnv points to the native method interface.

A new variable type introduced here is:

JDK1\_1InitArgs

This is a mandatory data type for JVM invocation from a program. Its value depends on the version and release level of the JDK that is used. For JDK 1.1, this value needs to be initialized to:

```
initArgs.version = 0x00010001;
```

This statement returns a default configuration for the JVM:

```
JNI_GetDefaultJavaVMInitArgs(&initArgs);
```

The initArgs must be set before calling this function, as shown in Figure 11-56.

```
JNI_GetDefaultJavaVMInitArgs(&initArgs);

#pragma convert(819)

myClasspath = malloc( strlen(initArgs.classpath) +
                      strlen(":/Javasamples/as400dir") + 1 );
strcpy ( myClasspath, initArgs.classpath );
strcat ( myClasspath,(":/Javasamples/as400dir");
initArgs.classpath = myClasspath;

JNI_CreateJavaVM(&myJVM, &myEnv, &initArgs);
```

Figure 11-56 Setting initArgs

As shown in Figure 11-56, the #pragma statement sets the language code page to ASCII, so any literal character string is treated as an ASCII string:

```
#pragma convert(819)
```

Next, memory is allocated to an already defined variable myClasspath of type char\*:

```
myClasspath = malloc( strlen(initArgs.classpath) +
                      strlen(":/Javasamples/as400dir") + 1 );
```

This variable is defined earlier. The length of the variable is set to the length of the String classpath of object initArgs + the length of user classpath + 1.

Next, we copy the classpath of the object initArgs to myClasspath:

```
strcpy(myClasspath, initArgs.classpath);
```

The next statement adds the integrated file system path where the user Java classes are stored to the myClasspath field:

```
strcat ( myClasspath,(":/Javasamples/as400dir");
```

We replace the current value of classpath with the value of myClasspath:

```
initArgs.classpath = myClasspath;
```

After setting the classpath in initArgs, this statement is executed:

```
JNI_CreateJavaVM(&myJVM, &myEnv, &initArgs);
```

This JNI function actually loads and initializes the JVM. The current thread becomes the main thread. Here are the variables:

- ▶ **&myJVM** is a pointer to the location where the resulting VM structure is placed.
- ▶ **&myEnv** is a pointer to the location where the JNI interface pointer for the main thread is placed.
- ▶ **&initArgs** is the configuration that you use to start the JVM.

As shown in Figure 11-57, this statement locates the Java program that is to be loaded:

```
myClass = (*myEnv) -> FindClass(myEnv, "ChangeStgJ");
```

In this example, we load the Java class, named ChangeStgJ. To load any other Java class that exists in your directory, simply change the name of the Java program.

```
myClass = (*myEnv) ->FindClass(myEnv, "ChangeStgJ");

mainID = (*myEnv) -> GetStaticMethodID(myEnv, myClass,
                                         "main", "(??(Ljava/lang/String;)V");

stringClass = (*myEnv) ->
    FindClass(myEnv, "java/lang/String");

args = (*myEnv) -> NewObjectArray(myEnv, 0, stringClass,0);

(*myEnv)->CallStaticVoidMethod(myEnv,myClass,mainID,args);

(*myJVM)->DestroyJavaVM(myJVM);

}
```

Figure 11-57 Loading the ChangeStgJ Java program

The GetStaticMethod JNI function loads the method whose name and signature has been specified in the argument:

```
mainID = (*myEnv) -> GetStaticMethod(myEnv, myClass,
                                         "main", "(??(Ljava/lang/String;)V");
```

**Note:** (?? is a replacement symbol being used instead of the bracket [.

In the next statement, the FindClass statement is used again. Instead of locating a program to call, the String class from the JDK-provided classes is loaded into StringClass.

```
StringClass = (*myEnv) ->
    FindClass(myEnv, "java/lang/String");
```

This statement defines a String array of zero length:

```
args = (*myEnv) -> NewObjectArray(myEnv, 0, stringClass, 0);
```

This is done to use a main method of any class, because a String array must be provided with the main method to accept parameter keyboard input. In the Java ChangeStgJ program, we do not accept any parameters from the keyboard, so this String needs to be initialized to zero.

This statement calls the main method of the myClass object using a String array with zero length:

```
(*myEnv) -> CallStaticVoidMethod(myEnv, myClass, mainID,args);
```

It actually starts the execution of the myClass program object. The myClass object that is invoked in this example is ChangeStgJ. The main ID is main.

After the invoked class completes processing, control is returned to the next statement, which is:

```
(*myJVM) -> DestroyJavaVM(myJVM);
```

This command unloads the JVM and reclaims its resources. Only the main thread can unload the JVM. The system waits until the main thread is the only remaining thread before it destroys the JVM.

To summarize, this example is different from the others because it uses the Invocation API. In this example, the C program loads the JVM, attaches any Java programs we want to run, and then executes the Java program. This program needs to be submitted in batch to run because it uses the multi-threaded capability of OS/400.

## Finding method signatures

The Java class disassembler, `javap`, is available as part of the JDK support. It disassembles class files and prints a human-readable version of the class specified. When it is executed with the `-s` option, it outputs class member declarations. This is useful for finding the signatures of Java methods. It can be used to determine signatures when calling Java methods from C or RPG.

For example, if we enter `javap -s ChangeStgJ`, we see the output displayed in Figure 11-58.

```
Compiled from ChangeStgJ.java
public synchronized class ChangeStgJ extends java.lang.Object
public java.lang.String s;
public native void setTheString();
public static void main(java.lang.String[]);
public ChangeStgJ();
static static {};
```

Figure 11-58 The `javap -s` output

It shows us the signature for the main method. It expects a String array parameter.





# Debugging Java programs on the iSeries server

This chapter examines how to debug Java programs that run on the iSeries server. There are two different ways that you can do this:

- ▶ You can use the OS/400 system debugger, which is described in 12.2, “Using the OS/400 system debugger” on page 479.
- ▶ You can use the IBM Distributed Debugger for iSeries, which is described in 12.3, “The IBM Distributed Debugger” on page 489.

We explain how to use both debug environments in this chapter. Both of these debuggers allow you to debug Java programs that run on the iSeries server. We use the RMI JDBC example described in 9.4, “RMI JDBC example” on page 355, to demonstrate debugging the host Java code.

## 12.1 Getting ready to debug

To debug Java programs on the iSeries server, you must:

- ▶ Compile the Java program with debug information.
- ▶ Compile with optimization level 10 or less.
- ▶ Place both the Java source and Java class file in the same directory in the integrated file system.

### 12.1.1 Compiling the code for debugging

Before you debug a Java program on the iSeries server, you need to compile the Java source with the debug option. You can do this by:

- ▶ Using iSeries the `javac` native Java compile command with the debug option.
- ▶ Exporting the Java source code and class file to the iSeries file system from VisualAge for Java 3.5 while including the debug attribute.

#### iSeries native compile

Use the `javac` command to compile the .java file with the `-g` option to obtain the debug information into the code, for example:

```
javac -g MyPackage\ MyClass.java
```

If you do not compile with the `-g` option, debugging with the Distributed Debugger and OS/400 native debugger is limited. Although you can step through the source code, you cannot inspect variables.

You may receive the message Identifier does not exist, when displaying variables in the OS/400 debugger. Or, you may receive a message similar to the one shown in Figure 12-1 from the Distributed Debugger. If you receive these messages, you probably did not compile your class with debug information. Compile it with debug information and try again.

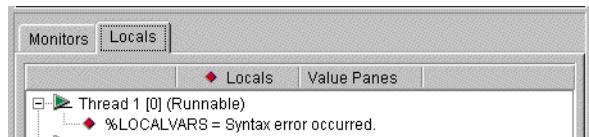


Figure 12-1 Debugger message

#### VisualAge for Java 3.5 compile

You can compile the iSeries server Java code in VisualAge for Java and export to it to the iSeries file system. See Figure 12-2.

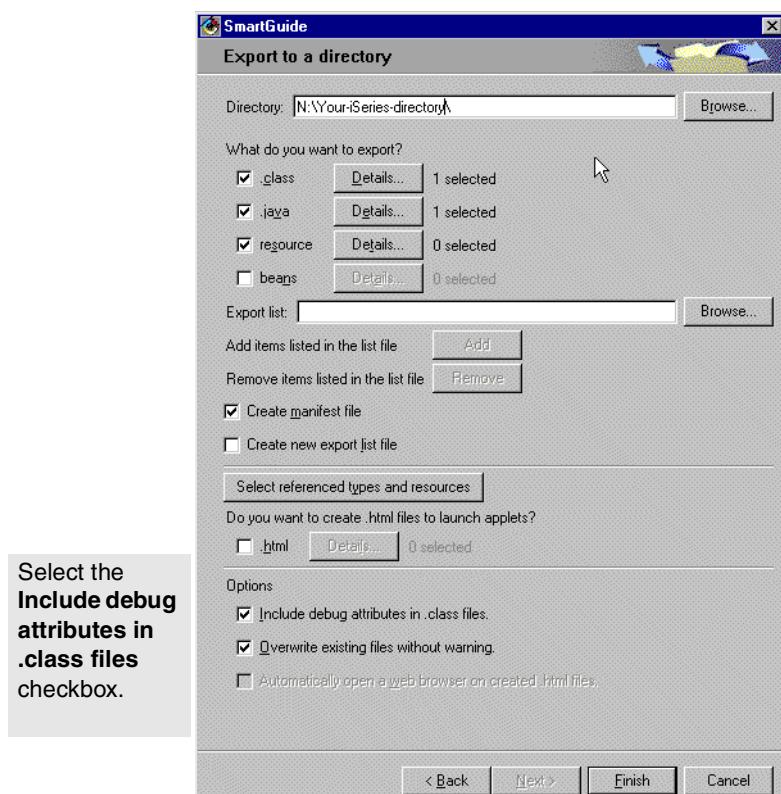


Figure 12-2 Export Debug attributes in the .class files

If you don't select the *Include debug attributes in .class files* option, you may receive the message Class file cannot be debugged, when displaying variables in the OS/400 debugger. Or, you may receive a message similar to the one shown in Figure 12-3 from the Distributed Debugger. Export it using this option and try again.

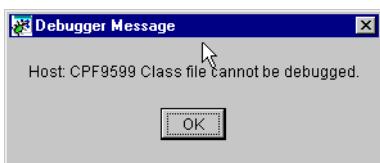


Figure 12-3 Debugger Message

You may have already compiled the program with the -g option and then used the CRTJVAPGM command or the Compile iSeries Java Class SmartGuide with an optimization level higher than 10. In this case, we recommend that you call the CRTJVAPGM command or the SmartGuide with an optimization level of 10. Then you can use the debugger capabilities.

## 12.2 Using the OS/400 system debugger

If you want to debug using the OS/400 debugger, start the Java program with the \*DEBUG option:

```
JAVA CLASS(packageName.className) OPTION(*DEBUG)
```

We are debugging the JDBCRmi class, which is found in the JDBCMIExample package:

```
JAVA CLASS(JDBCMIExample.JDBCRmi) OPTION(*DEBUG)
```

The Display Module Source display is shown in Figure 12-4.

**Important:** To debug a Java program on the iSeries server, you must set up the Java environment properly. In this case, we need the CLASSPATH environment variable set properly so our classes can be found. For information on how to do this, see 2.2, “Setting up the environment on the iSeries server” on page 44.

Display Module Source

```
Class file name: JDBCMIExample.JDBCRmi
1 package JDBCMIExample;
2
3 /**
4  * This class was generated by a SmartGuide.
5  *
6  */
7
8 import com.ibm.as400.access.*;
9 import java.math.*;
10 import java.sql.*;
11 import java.util.*;
12 import java.text.*;
13 import java.rmi.*;
14
15 import java.rmi.server.*;
```

More...

Debug . . .

F3=End program F6=Add/Clear breakpoint F10=Step F11=Display variable  
F12=Resume F17=Watch variable F18=Work with watch F24=More keys

Figure 12-4 Display Module Source

The following function keys are supported:

- ▶ F3 ends the program.
- ▶ F6 toggles breakpoints. The program source is displayed when the breakpoint is encountered.
- ▶ F10 steps through the program.
- ▶ F11 displays the value of the variable under the cursor.
- ▶ F12 resumes or runs the program.
- ▶ F14 lets you work with the list of classes that you use in your program.
- ▶ F17 and F18 are not available in Java.
- ▶ F22 lets you step into a method.

### 12.2.1 Setting breakpoints

To set a breakpoint at a particular line, use the Page Down key to move through the source code. When you find the line where you want to set the breakpoint, press the F6 key. The F6 key is a toggle key that sets the breakpoint to the opposite of its current setting. In the example shown in Figure 12-5, we set the breakpoint at line 95.

```

Display Module Source

Class file name: JDBCExample.JDBC
91 * This method was created by a SmartGuide.
92 */
93 public Item getItem (String anItem) throws RemoteException
94 {
95 Item theItem = new Item(anItem);
96 try
97 {
98 System.out.println("getItem: "+anItem);
99 java.sql.ResultSet rs = null;
100 psSingleRecord.setInt(1, Integer.parseInt(anItem));
101 rs = psSingleRecord.executeQuery();
102 if (rs.next()) {
103 System.out.println(" record found");
104 theItem.setItemDesc(rs.getString("PARTDS"));
105 theItem.setItemQty(rs.getInt("PARTQY"));
106 }
107 }
108 catch (Exception e)
109 {
110 System.out.println("Exception caught: " + e.getMessage());
111 }
112 finally
113 {
114 if (rs != null)
115 rs.close();
116 }
117 }
118 }

More...

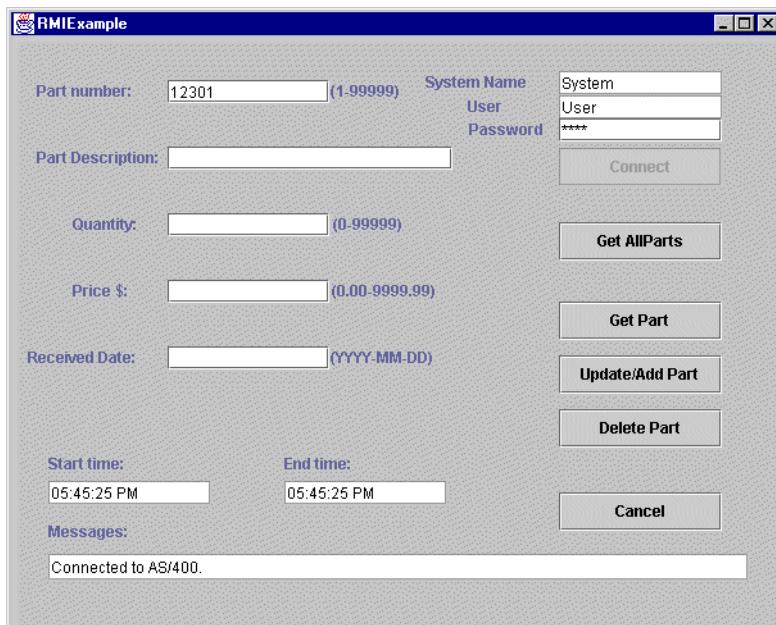
Debug . . .

F3=End program F6=Add/Clear breakpoint F10=Step F11=Display variable
F12=Resume F17=Watch variable F18=Work with watch F24=More keys
Breakpoint added to line 95.

```

*Figure 12-5 Setting a breakpoint*

Now, we are ready to debug the program. In this example, the server RMI program is invoked from the RMI client program. As shown in Figure 12-6, we start the client program, connect to the server, and attempt to retrieve a part record from the database. When we click the Get Part button, the host program is called.



*Figure 12-6 Starting the client RMI program*

When we call the host program, it stops at line 95 before running it (Figure 12-7 on page 482).

```

Display Module Source
Current thread: 000000A7     Stopped thread: 000000A7
Class file name: JDBCExample.JDBCExample
91   * This method was created by a SmartGuide.
92   */
93 public Item getItem (String anItem) throws RemoteException
94 {
95 Item theItem = new Item(anItem);
96 try
97 {
98 System.out.println("getItem: "+anItem);
99 java.sql.ResultSet rs = null;
100 psSingleRecord.setInt(1, Integer.parseInt(anItem));
101 rs = psSingleRecord.executeQuery();
102 if (rs.next()) {
103 System.out.println(" record found");
104 theItem.setItemDesc(rs.getString("PARTDS"));
105 theItem.setItemQty(rs.getInt("PARTQY"));
More...
Debug . . .

F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable  F18=Work with watch  F24=More keys
Breakpoint at line 95 in thread 000000A7

```

*Figure 12-7 Stopping at a breakpoint*

## 12.2.2 Displaying variables

The debugger allows you to display program variables. To display a variable, on the command line, enter:

EVAL VARIABLENAME

Figure 12-8 shows how to evaluate a String variable called anItem.

```

Display Module Source
Current thread: 000000A7 Stopped thread: 000000A7
Class file name: JDBCExample.JDBC
91 * This method was created by a SmartGuide.
92 */
93 public Item getItem (String anItem) throws RemoteException
94 {
95 Item theItem = new Item(anItem);
96 try
97 {
98 System.out.println("getItem: "+anItem);
99 java.sql.ResultSet rs = null;
100 psSingleRecord.setInt(1, Integer.parseInt(anItem));
101 rs = psSingleRecord.executeQuery();
102 if (rs.next()) {
103 System.out.println(" record found");
104 theItem.setItemDesc(rs.getString("PARTDS"));
105 theItem.setItemQty(rs.getInt("PARTQY"));
More...
Debug . . . EVAL anItem

F3=End program F6=Add/Clear breakpoint F10=Step F11=Display variable
F12=Resume F17=Watch variable F18=Work with watch F24=More keys
anItem = 12321

```

*Figure 12-8 Using the EVAL function*

EVAL allows you to display:

- ▶ An instance of a class
- ▶ A data member
- ▶ Strings
- ▶ Integers
- ▶ Arrays
- ▶ An Array element

You can change variables, except for:

- ▶ Instances of classes
- ▶ Strings
- ▶ Arrays

You can also use EVAL to display the number of elements in an array (for example, enter arrayname.length). You can also compare objects for equality.

If the variable is an object, a special Evaluate Expression window is shown. In Figure 12-9 on page 484, we evaluate an object called theItem.

```

Display Module Source
Current thread: 000000A7 Stopped thread: 000000A7
Class file name: JDBCExample.JDBCRmi
91 * This method was created by a SmartGuide.
92 */
93 public Item getItem (String anItem) throws RemoteException
94 {
95 Item theItem = new Item(anItem);
96 try
97 {
98 System.out.println("getItem: "+anItem);
99 java.sql.ResultSet rs = null;
100 psSingleRecord.setInt(1, Integer.parseInt(anItem));
101 rs = psSingleRecord.executeQuery();
102 if (rs.next()) {
103 System.out.println(" record found");
104 theItem.setItemDesc(rs.getString("PARTDS"));
105 theItem.setItemQty(rs.getInt("PARTQY"));
More...
Debug . . . EVAL theItem

F3=End program F6=Add/Clear breakpoint F10=Step F11=Display variable
F12=Resume F17=Watch variable F18=Work with watch F24=More keys

```

*Figure 12-9 Using the EVAL function on an object*

Figure 12-10 shows the object Evaluate Expression display.

```

Evaluate Expression

Previous debug expressions

-----> THREAD 000000A7 <-----
> EVAL anItem
anItem = 12321
> EVAL theItem
theItem = JDBCExample/Item:FDB7BCA6B401B300
theItem.ItemId = java/lang/StringBuffer:C70E0E2FD306E670
theItem.ItemDesc = null
theItem.ItemPrice = null
theItem.ItemDate = null
theItem.entryArray = ARR:EB0B19947601A000
theItem.ItemQuantity = 0
theItem.index = 0
More...
Debug . . .

F3=Exit F9=Retrieve F12=Cancel F16=Repeat find F19=Left F20=Right
F21=Command entry F23=Display output

```

*Figure 12-10 Evaluating an expression for an object*

You can use the EVAL function to set variables or to test objects for equality. Figure 12-11 shows how to compare the value of the theItem object ItemQuantity variable to 43. In this case, it is false. Next, we set it to 43 and then compare it again. This time it is true. Then, we use the EVAL function to display the entire theItem object.

**Note:** We use == for comparison and = to set equal to, which is the same as if we were writing Java code.

```
Evaluate Expression

Previous debug expressions
> EVAL theItem
theItem = JDBCExample/Item:FDB7BCA6B401B300
theItem.ItemId = java/lang/StringBuffer:C70E0E2FD306E670
theItem.ItemDesc = null
theItem.ItemPrice = null
theItem.ItemDate = null
theItem.entryArray = ARR:EB0B19947601A000
theItem.ItemQuantity = 0
theItem.index = 0
> EVAL theItem.ItemQuantity == 43
theItem.ItemQuantity == 43 = FALSE
> EVAL theItem.ItemQuantity == 0
theItem.ItemQuantity == 0 = TRUE
> EVAL theItem.ItemQuantity = 43
theItem.ItemQuantity = 43 = 43
> EVAL theItem.ItemQuantity == 43
theItem.ItemQuantity == 43 = TRUE
> EVAL theItem
theItem = JDBCExample/Item:FDB7BCA6B401B300
theItem.ItemId = java/lang/StringBuffer:C70E0E2FD306E670
theItem.ItemDesc = null
theItem.ItemPrice = null
theItem.ItemDate = null
theItem.entryArray = ARR:EB0B19947601A000
theItem.ItemQuantity = 43
theItem.index = 0
Bottom
Debug . . .
F3=Exit F9=Retrieve F12=Cancel F16=Repeat find F19=Left F20=Right
F21=Command entry F23=Display output
```

Figure 12-11 Setting a variable

### 12.2.3 Work with module list

The F14 key allows you to add other programs to the debug session. When you select it, the Work with Module List display is shown. You can use this display to start debug sessions for:

- ▶ Classes that are called by the initial class
- ▶ \*SRVPGM modules that contain native methods
- ▶ \*PGM modules that are called by native methods

The Work with Module List display is shown in Figure 12-12 on page 486. You should specify the Library parameter with \*LIBL for \*CLASS type modules.

```

Work with Module List
System: [REDACTED]

Type options, press enter.
  1=Add program   4=Remove program   5=Display module source
  8=Work with module breakpoints

Opt     Program/module      Library      Type
        *LIBL             *PGM
<Rmi.JDBCRmi           *CLASS       Selected

Command

====>
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
F22=Display class file name

```

*Figure 12-12 Work with Module List*

#### 12.2.4 Debugging from another terminal session

You may choose to debug the Java program from an iSeries session other than the one in which the Java program is running. This is convenient if you want to watch both the Java program window and debug display simultaneously. You can also use this technique to remotely debug a Java program. To do this, you must complete these steps:

1. Start the Java program that you want to debug. The Java program must stop, so you can access its source with the debugger and set the appropriate breakpoints that you want.
2. Sign on to an iSeries 5250 emulation session.
3. Determine the job information for the Java program that you want to debug. The Work with Active Jobs (WRKACTJOB) command is useful for this.
4. Start a service job using the Start Service Job (**STRSRVJOB**) command. This command requires job information for the job it is to service.
- Note:** The user must be authorized to the STRSRVJOB command and the STRDBG command.
5. Issue the Start Debug (**STRDBG**) command for the Java class that you want to debug. The source for the Java program displays, and you can debug it as described previously.

Now, we go through the steps to debug a Java program on iSeries from an independent display session. First, we sign on and run the Work with Active Jobs (WRKACTJOB) command. We look for the BCI job for the Java program. In this case, we find it as the QJVACMDSRV job for user MAATTA. We use option 5 to work with the job (Figure 12-13).

```

Work with Active Jobs
CPU %: .0     Elapsed time: 00:00:00     Active jobs: 230

Type options, press Enter.
2=Change   3=Hold   4=End   5=Work with   6=Release   7=Display message
8=Work with spooled files 13=Disconnect ...

Opt Subsystem/Job User      Type CPU % Function      Status
5      DSP04        WBL       INT  .0  PGM-QCMD      DSPW
      QJVACMDSRV  MAATTA    BCI  .0
      QPADEV0007  MAATTA    INT  .0  CMD-JAVA      TIMW
      QPADEV0009  MAATTA    INT  .0  CMD-QSH       DEQW
      QPADEV0010  MAATTA    INT  .0  CMD-WRKACTJOB  RUN
      QPADEV0016  JAREK     INT  .0  CMD-WRKOBJPDM  DSPW
      QZSHSH       MAATTA    BCI  .0
      QSERVER      QSYS      SBS  .0
      QPWFSERVSD  QUSER     BCH  .0
                                         SELW

Parameters or command

====>
F3=Exit   F4=Prompt      F5=Refresh   F10=Restart statistics
F11=Display elapsed data F12=Cancel   F14=Include   F24=More keys

```

*Figure 12-13 Work with Active Jobs*

Then, the Work with Job display (Figure 12-14) appears. It shows the Job Name, the User, and the Job Number information that we need to start a service job.

```

Work with Job

Job: QJVACMDSRV      User: MAATTA      Number: 061072

Select one of the following:

1. Display job status attributes
2. Display job definition attributes
3. Display job run attributes, if active
4. Work with spooled files

10. Display job log, if active or on job queue
11. Display call stack, if active
12. Work with locks, if active
13. Display library list, if active
14. Display open files, if active
15. Display file overrides, if active
16. Display commitment control status, if active

Selection or command

====>

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel

```

*Figure 12-14 Work with Job*

Use the STRSRVJOB command prompt display to start a service job for the Java program. See Figure 12-15 on page 488.

```

Start Service Job (STRSRVJOB)

Type choices, press Enter.

Job name . . . . . > QJVACMDSRV      Name
User . . . . . > MAATTA        Name
Number . . . . . > 061072      000000-999999

F3=Exit   F4=Prompt   F5=Refresh   F10=Additional parameters   F12=Cancel
F13=How to use this display   F24=More keys

```

Figure 12-15 Start Service Job (STRSRVJOB)

The final step is to start a debug session for the job using the STRDBG command. We use the CLASS parameter to enter the Java class to debug. It is entered in the format *packageName.className*.

STRDBG CLASS(JDBCRmi.JDBCRmi)

The source for the Java program displays and you can debug it by setting breakpoints, stepping though code, or displaying or modifying variables as discussed previously. See Figure 12-16.

```

Display Module Source

Class file name: JDBCRmi.JDBCRmi
1
2 package JDBCRemote;
3
4 /**
5  * This class was generated by a SmartGuide.
6  *
7  */
8
9 import com.ibm.as400.access.*;
10 import java.math.*; // for BigDecimal class
11 import java.sql.*; // for JDBC classes
12 import java.util.*; // for Properties class
13 import java.text.*; // for DateFormat class
14 import java.rmi.*; // for Remote Method Invocation
15 import java.rmi.registry.*;

Debug . . .

F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More key

```

Figure 12-16 Display Module Source

## 12.3 The IBM Distributed Debugger

The Distributed Debugger allows you to debug Java programs that run on the iSeries server. The debugger user interface runs on the workstation platform. It allows you to control the flow of the program, set breakpoints, and work with program variables in a more graphical manner than the OS/400 debugger. The Distributed Debugger can debug both client and server Java code.

Before you can use the debugger, the debug server must be started on the iSeries server. To start the debug server, enter the Start Debug Server (STRDBGSRV) command on an iSeries command line and press Enter.

**Attention:** The debug server needs to be started only once for the iSeries Server on which you plan to debug your application.

If the debug server was already started previously, you see the message Debug server router function already active, when you issue the STRDBGSRV command.

To debug an iSeries Java program, you need to compile it using the -g option or by using the *Include debug attributes in .class files* option in VisualAge for Java. If you compiled the program and used a higher optimization level higher than 10, recompile with level 10 for best results. Use the command:

```
CRTJVAPGM CLSF(xxxxxx) OPTIMIZE(10)
```

**Note:** Before using the Distributed Debugger, make sure that the following PTFs are installed:

- ▶ MF26612 and SI01218 for V5R1
- ▶ MF26607 and SF65598 for V4R5'

### 12.3.1 Starting the Distributed Debugger

To start the Distributed Debugger, follow these steps:

1. Launch the Distributed Debugger from the Windows program start menu. To do so, select **Programs-> IBM VisualAge for Java for Windows-> IBM Distributed Debugger-> IBM Distributed Debugger**.
2. Click **Cancel** on the Load Program window.
3. From the Distributed Debugger window, select **File-> Attach**. As shown in Figure 12-17 on page 490, you now see the Attach dialog.

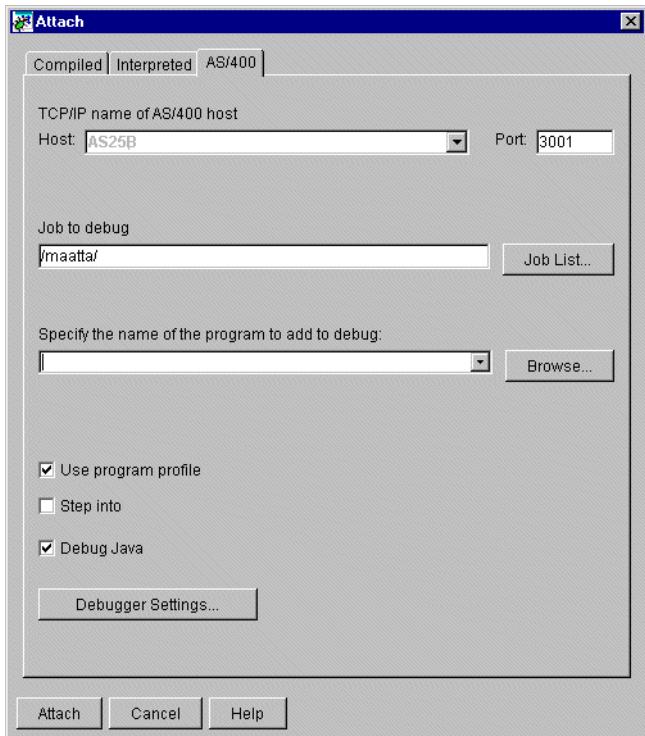


Figure 12-17 Attaching the debugger to a program

4. In the Attach window, enter the host name of the system where the program that you want to debug is running.
5. You need to attach the debugger to the Java program running on the server. You can use the Job List button to help you find the job to attach to. You can use the Job to Debug entry field to narrow the list of jobs displayed. In Figure 12-17, we use the user ID to limit the jobs displayed. Click the **Job List** button to display the list of jobs.

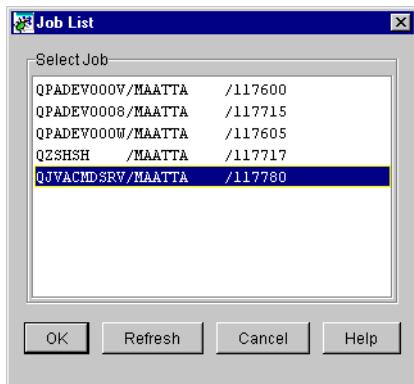


Figure 12-18 Displaying the job list

6. As shown in Figure 12-19, select the job you want to attach to, and click **OK** to return to the Attach window.

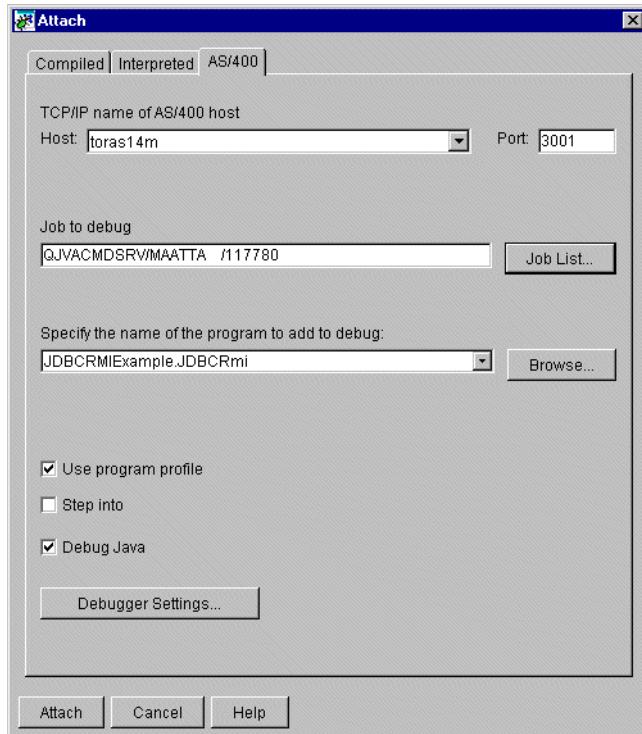


Figure 12-19 Attaching the debugger to the server job

7. Click **Attach** to attach the debugger to the server job. The Debugger Logon prompt shown in Figure 12-20 appears. Enter your user ID and password. Then click **OK**.

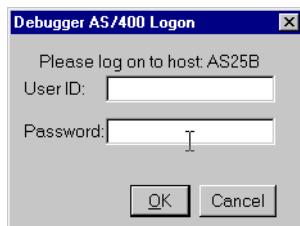


Figure 12-20 Debugger AS/400 (iSeries) Logon

8. After that, you see the main Distributed Debugger window. Select the **Programs** tab, expand the class (JDBCRMIEExample/JDBCRmi), and select the **getItem** method from the list. You then see the source Java code as shown in Figure 12-21 on page 492.

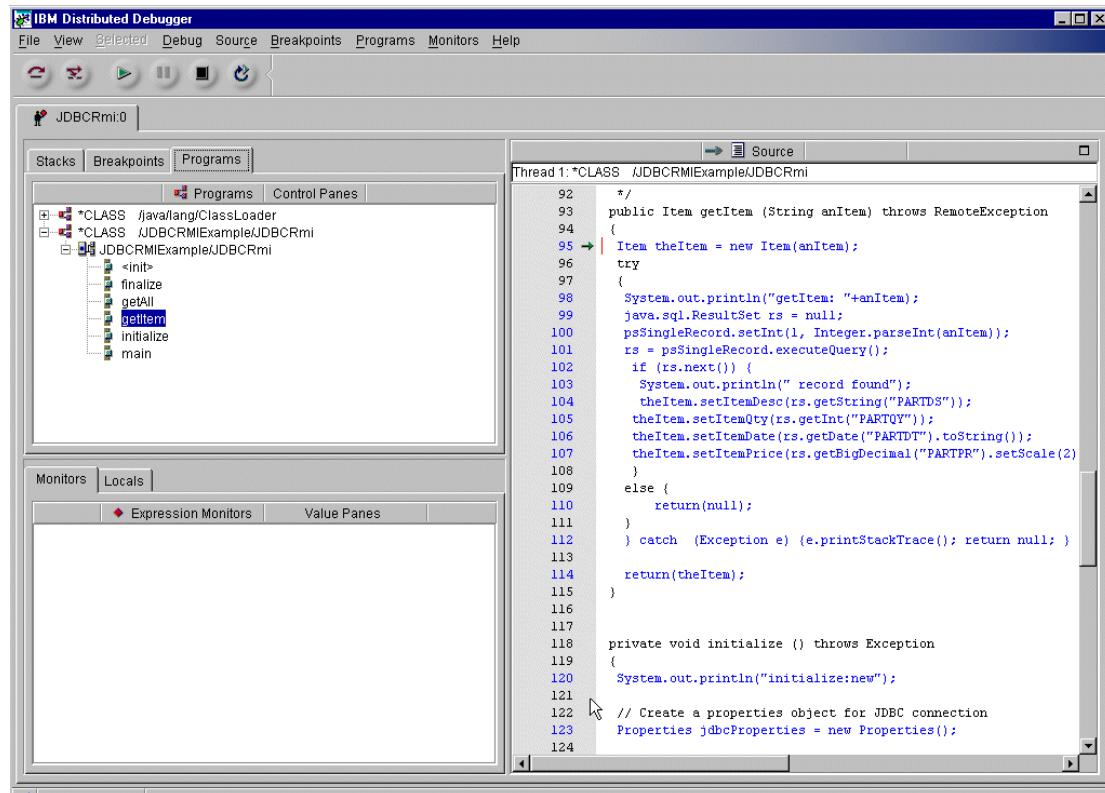


Figure 12-21 Distributed Debugger initial window

### 12.3.2 Debugging an iSeries Java program

To debug iSeries Java programs, follow these steps:

1. In the source panel of the main debug window, you can set breakpoints by double-clicking the line number where you want the breakpoint. We add a breakpoint in line 95 as shown in Figure 12-22.

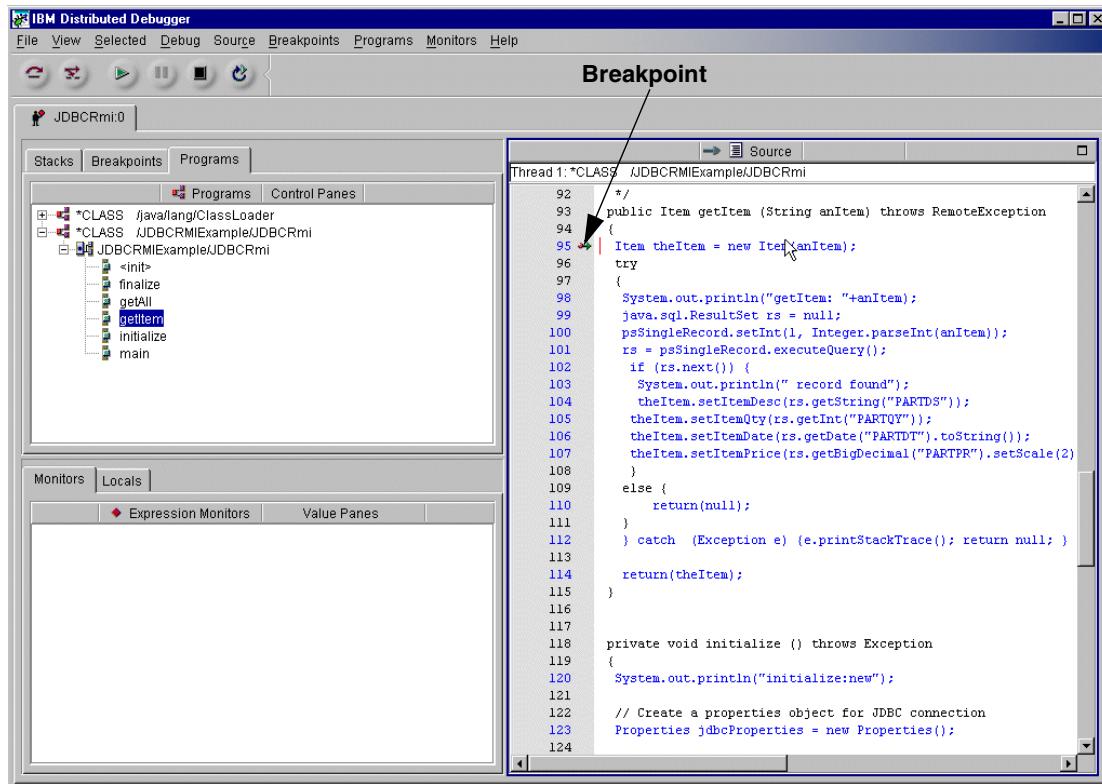


Figure 12-22 Debug source window

- Start debugging by running the application. You can click the **Run** button (or use the keyboard shortcut F5). See Figure 12-23.



Figure 12-23 Distributed Debugger shortcut

- Click **OK** to the debugger message (Figure 12-24) that tells you to start your application.

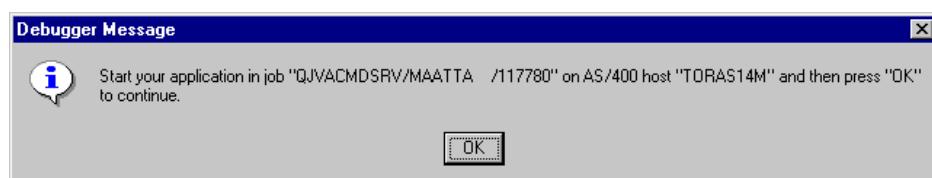


Figure 12-24 Debugger message

- Use VisualAge for Java to start the client application. See Figure 12-25 on page 494.

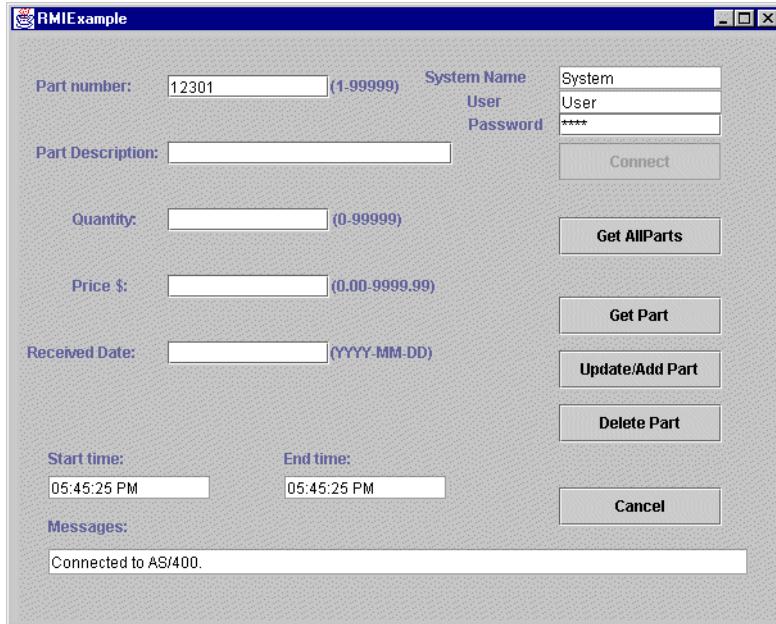


Figure 12-25 Running the client application

- After connecting to the iSeries server, click the **Get Part** button in the client application. The server application stops at the breakpoint that was set earlier. Click F8 to execute the line 95 code. The local variables are updated as you step through the program. You can display the theItem and anItem variables values as shown in Figure 12-26.

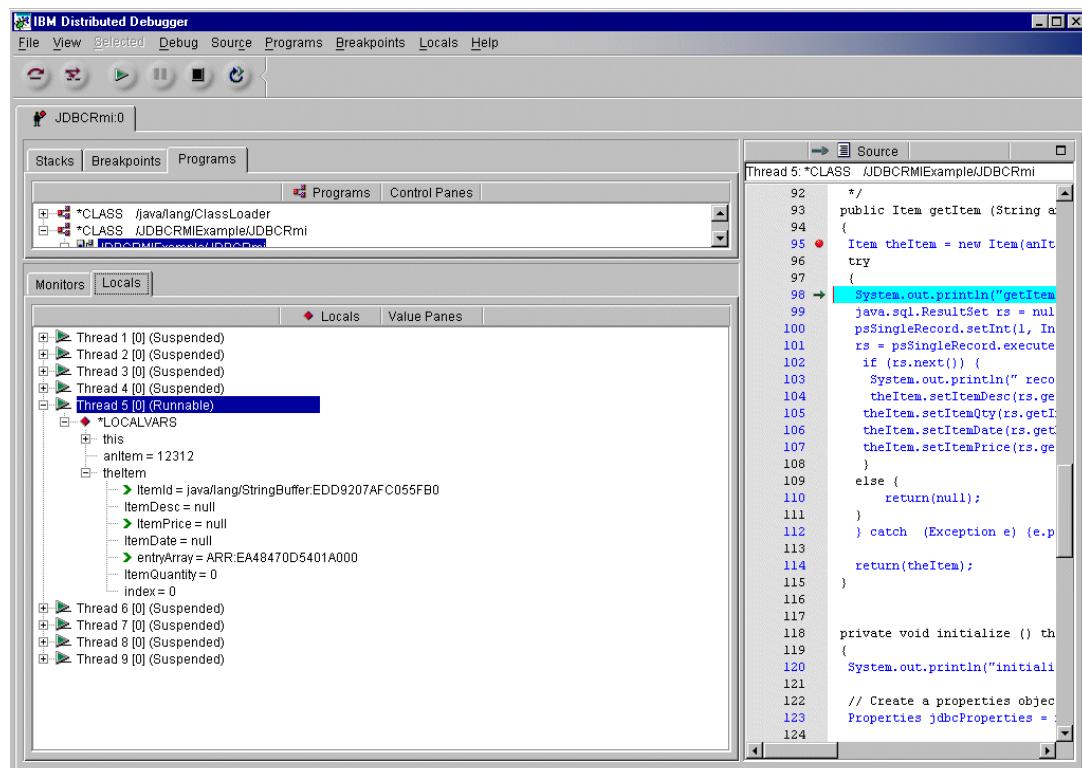


Figure 12-26 Distributed Debugger window

If you need to change any of the local variables, simply double-click the value of the variable. Make the change, and press the Enter key.

### 12.3.3 Controlling the Distributed Debugger session

For basic debugging, you can use the Distributed Debugger menu and icons (Figure 12-27) to control a debug session. These menu items and icons cause debugger commands to run.

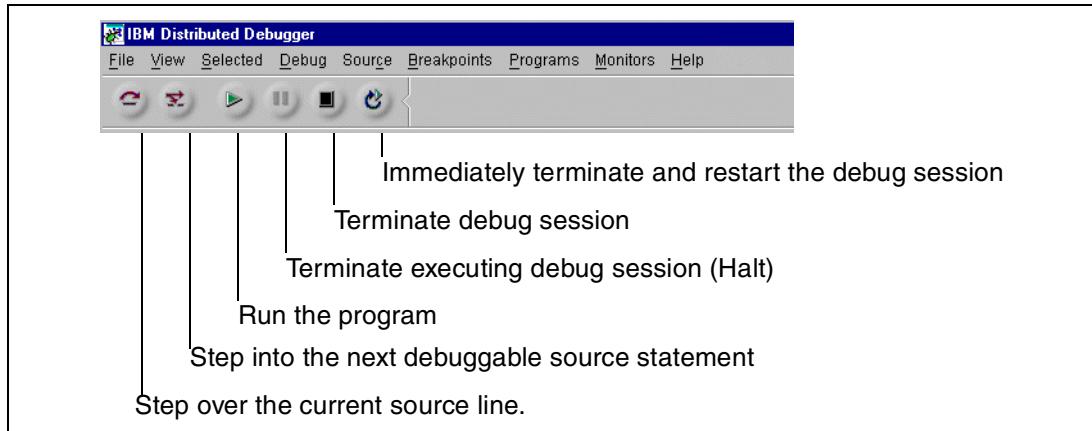


Figure 12-27 Debugger control menu and icons

The following list describes some key session control menu items and icons and explains how to use them:

► **Step Over**

Step Over runs the current line in the program. If you step over a method that contains a breakpoint, execution stops at the breakpoint.

► **Step Into**

Step Into runs the current line in the program. If the current line is a method call, and debug information is available for any methods called, execution halts at the first statement for which debug information is available. If no debug information is available, execution halts after the method call.

For example, A calls B and A is compiled without debug information, while B is compiled with debug information. After a Step Debug from a call to A, execution stops at the first executable statement of B. The debugger steps over any methods for which debugging data is not available.

► **Run**

Run executes the program. The debugger starts executing where it last stopped (for example, at a breakpoint). Or if you have not yet run the program in this session, it starts at the beginning of the program. When you select Run, execution continues until one of these conditions occur:

- End of program is reached
- An active breakpoint is hit
- A specific line number is reached
- An exception occurs.

► **Restart**

Depending on the debug session startup mode or language under debug, restarting a program will cause the following situations:

- If you attached the debugger to a job when you started the debug session, a message prompts you to call the program again in your iSeries job before the debug session restarts.
- If you attached the debugger to a job without specifying a program name and then ran the program to termination, you can restart the debugger for additional debug sessions.
- If you loaded the program into the debugger when you started the debug session, the program will restart automatically when you click Restart.

**Note:** If you restart an interactive program that produces 5250 screen output, the debugger will run the program to termination before restarting it. You will then have less time than is necessary to dismiss the 5250 screen before the program restarts. Dismissing the 5250 screen will cause termination of the restarted program. When debugging such a program, running to the last statement in the program before restarting will allow for better restart behavior.

► **Terminating a debug session (iSeries)**

To terminate a debug session when the program is executing on the iSeries server, click the **Halt** push button. The Halt push button is available when the program being debugged on the iSeries is executing. If the program stops, then the Halt push button is not available.

When the debugger has control of the program, you can terminate the debug session and exit the debugger by choosing one of the following options:

- Click the **Terminate (Halt)** toolbar button.
- Select **File-> Exit** from the Distributed Debugger window.
- In the debugger window, use your windowing system's technique for closing the window. For example, double-click the upper left corner of the window or press Alt+F4. You can also single click the upper right corner.



# Deployment considerations and tools

This chapter introduces some deployment considerations and the tools available to enable you to deploy finished Java applications more effectively.

The items covered in this chapter are:

- ▶ How to Install and update the IBM Toolbox for Java functions
- ▶ How to reduce the size of Java archive files (.jar and .zip files)
- ▶ How to encrypt data transmissions between a client and server

## 13.1 The IBM Toolbox for Java installation and update

The IBM Toolbox for Java classes can be referenced at their location in the integrated file system on the iSeries server. Because program temporary fixes (PTFs) are applied to this location, Java programs that access these classes directly on the iSeries server automatically receive these updates. Accessing the classes from the iSeries server does not work for every situation, specifically those listed here:

- ▶ If a low-speed communication link connects the iSeries server and the client, the performance of loading the classes from the iSeries may be unacceptable.
- ▶ If the Java applications use the CLASSPATH environment variable to access the classes on the iSeries server file system, you need a network drive to access the iSeries integrated file system. Software, such as Client Access Express, is required to support network drives. It may not be possible to install this software in the client.

In those cases, installing the classes on the client is a better solution.

The AS400ToolboxInstaller class provides client installation and update functions to manage IBM Toolbox for Java classes when they reside on a client.

You can invoke the AS400ToolboxInstaller object:

- ▶ From within your program
- ▶ From a command line

If your Java program uses IBM Toolbox for Java functions, you can include the AS400ToolboxInstaller class as a part of your program. When the Java program is installed or first run, it can use the AS400ToolboxInstaller class to install the IBM Toolbox for Java classes on the client. When the Java program is restarted, it can use the AS400ToolboxInstaller to update the classes on the client.

**Note:** If you are using the V3R2 or V3R2M1 version of the IBM Toolbox for Java and you want to upgrade to V4R2 or a later version, you must use a V4R2 or later level of the AS400ToolboxInstaller class. You must use this level to ensure that machine readable information (MRI), stored in .property files in earlier releases of the IBM Toolbox for Java, is properly replaced by .class files used in later releases.

The AS400ToolboxInstaller class copies files to the client's local file system. This class may not work in an applet; many browsers do not allow a Java program to write to the local file system.

### 13.1.1 Embedding the AS400ToolboxInstaller class in your program

The AS400ToolboxInstaller class provides the APIs that are necessary to install, uninstall, and update IBM Toolbox for Java classes from within the program on the client.

Use the `install()` method to install or update the IBM Toolbox for Java classes. To install or update, provide the source and target path, as well as the name of the package of classes in your Java program. The source URL points to the location of the control files on the server. The directory structure is copied from the server to the client.

The `install()` method only copies files; it does not update the CLASSPATH environment variable. If the `install()` method is successful, the Java program can call the `getClasspathAdditions()` method to determine what must be added to the CLASSPATH environment variable.

The following example shows how to use the AS400ToolboxInstaller class to install files from an iSeries named "mySystem" to the directory "jt400" on drive d:. It also shows how to determine what must be added to the CLASSPATH environment variable:

```
URL sourceURL = new URL("http://mySystem.myCompany.com/QIBM/ProdData/HTTP/Public/jt400/");

if (AS400ToolboxInstaller.install( "ACCESS", "d:\\jt400", sourceURL)) {
    Vector additions = AS400ToolboxInstaller.getClasspathAdditions();
    if (additions.size() > 0) {
        // ... Process each classpath addition
    }
}

// ... Else no updates were needed.
```

Use the `isInstalled()` method to determine if the IBM Toolbox for Java classes are already installed on the client. Using the `isInstalled()` method allows you to determine if you want to complete the installation now or postpone it to a more convenient time.

The `install()` method both installs and updates files on the client. A Java program can call the `isUpdateNeeded()` method to determine if an update is needed before calling the `install()` method.

Use the `unInstall()` method to remove the IBM Toolbox for Java classes from the client. The `unInstall` method only removes files; the CLASSPATH environment variable is not changed. Call the `getClasspathRemovals()` method to determine what can be removed from the CLASSPATH environment variable.

### 13.1.2 Running the AS400ToolboxInstaller class from the command line

The AS400ToolboxInstaller class can be used as a stand-alone program and run from the command line. Running the AS400ToolboxInstaller from the command line means you do not have to write a program. Instead, you run it as a Java application to install, uninstall, or update the IBM Toolbox for Java classes.

When you specify the appropriate install, uninstall, or compare option, you invoke the AS400ToolboxInstaller class with the following command:

```
java utilities.AS400ToolboxInstaller [options]
```

The options are:

► **-install**

Indicates that the specified packages are to be installed. If the -package option is not specified, all packages will be installed. The source and target options must be specified when using this option. The -install option may be abbreviated to -i.

► **-uninstall**

Indicates that the specified packages are to be removed. The package and target options must be specified when using this option. The -uninstall option may be abbreviated to -u.

► **-compare**

Indicates that the source package will be compared with the target package to determine if an update is needed. The package, source, and target options must be specified when using this option. The -compare option may be abbreviated to -c.

► **-package package1[,package2[, ...]]**

Specifies the package to install, compare, or uninstall. If -package is specified, at least one package name must be specified. The -package option may be abbreviated to -p.

- ▶ **-source *sourceURL***  
Specifies the location of the source files. The HTTP server is used to access the files. The system name or the fully-qualified URL may be specified. If the system name is specified, it is automatically converted to the URL where the licensed program installed the files. For example, if mySystem is specified, <http://mySystem/QIBM/ProdData/HTTP/Public/jt400/> will be used. The -source option may be abbreviated to -s.
- ▶ **-target *targetDirectory***  
Specifies the fully-qualified path name of where to store the files. The -target option may be abbreviated to -t.
- ▶ **-prompt**  
Specifies that the user will be prompted before updating the packages on the workstation. If not specified, the packages will be updated. The -prompt option may be abbreviated to -pr.
- ▶ **-?**  
Displays the help text.
- ▶ **-help**  
Displays the help text. The -help parameter may be abbreviated to -h.

Options are also available to install the entire Toolbox or only certain functions. For example, an option exists to install just the proxy classes of the IBM Toolbox for Java.

## 13.2 Java archive files

Reducing the Java archive size is important for two reasons:

- ▶ A smaller archive is faster for downloaded to a client.
- ▶ A smaller archive is faster to search (this is usually a minor performance improvement).

Reducing the size of a Java archive can be important when the archive is downloaded across the Internet or through a slow WAN connection. The current jt400.jar file is 3.2 MB. With a slow modem or congested Internet route, downloading this file from a Web server can be time consuming.

To help overcome this problem, two tools are available with the IBM Toolbox for Java Modification 2 or later. The following classes are found in the </QIBM/ProdData/Http/Public/jt400/utilities> integrated file system directory and perform similar functions:

- ▶ **JarMaker**: A general purpose archive tool, primarily used to extract classes from JAR files and re-package them as new JAR files.
- ▶ **AS400ToolboxJarMaker**: An extension to JarMaker, specifically for the IBM Toolbox for Java JAR files. It can be used to extract specified iSeries components from the jt400.jar file to produce a new JAR file containing only the selected components.

### 13.2.1 JarMaker

The JarMaker class is used to generate a smaller (and, therefore, faster loading) JAR or ZIP file from a larger one. In addition, you can also use JarMaker to:

- ▶ Extract desired files from a JAR or ZIP file.
- ▶ Split a JAR or ZIP file into smaller JAR or ZIP files.

You can use JarMaker in a program, or run it from a command line:

```
java utilities.JarMaker [ options ]
```

You must specify at least one of the following options:

- ▶ **-requiredFile**
- ▶ **-additionalFile**
- ▶ **-package**
- ▶ **-extract**
- ▶ **-split**

If the following options are specified multiple times in a single command string, only the final specification applies:

- ▶ **-source**
- ▶ **-destination**
- ▶ **-additionalFilesDirectory**
- ▶ **-extract**
- ▶ **-split**

Other options have a cumulative effect when you specify them multiple times in a single command string.

## Options

These options control what JarMaker will do:

- ▶ **-source sourceJarFile**

Specifies the source JAR or ZIP file from which to derive the destination JAR or ZIP file. If a relative path is specified, the path is assumed to be relative to the current directory. If this option is specified as the first positional argument, the tag (-source) is optional. The -source option may be abbreviated to -s.

- ▶ **-destination destinationJarFile**

Specifies the destination JAR or ZIP file, which will contain the desired subset of the files in the source JAR or ZIP file. If a pathname is not specified, the file is created in the current directory. The -destination option may be abbreviated to -d. The default name is generated by appending "Small" to the source file name. For example, if the source file is myfile.jar, then the default destination file would be myfileSmall.jar.

- ▶ **-requiredFile jarEntry1[,jarEntry2[...]]**

The files in the source JAR or ZIP file that are to be copied to the destination. Entries are separated by commas (no spaces). The specified files, along with all of their dependencies, will be considered as required. Files are specified in the JAR entry name syntax, such as com/ibm/as400/access/DataQueue.class. The -requiredFile option may be abbreviated to -rf.

- ▶ **-additionalFile file1[,file2[...]]**

Specifies additional files (not included in the source JAR or ZIP file) that are to be copied to the destination. Entries are separated by commas (no spaces). Files are specified by either their absolute path or their path relative to the current directory. The specified files will be included, regardless of the settings of other options. The -additionalFile option may be abbreviated to -af.

► **-additionalFilesDirectory baseDirectory**

Specifies the base directory for additional files. This should be the parent directory of the directory where the package path starts. For example, if the foo.class file in the com.ibm.mypackage package is located in the C:\dir1\subdir2\com\ibm\mypackage\ directory, then specify a base directory of C:\dir1\subdir2. The -additionalFilesDirectory option may be abbreviated to -afd. The default is the current directory.

► **-package package1[,package2[...]]**

The packages that are required. Entries are separated by commas (no spaces). The -package option may be abbreviated to -p. Package names are specified in standard syntax, such as com.ibm.component.

**Note:** The specified packages are simply included in the output. No additional dependency analysis is done on the files in a package, unless they are explicitly specified as required files.

► **-extract [baseDirectory]**

Extracts the desired entries of the source JAR or ZIP file into the specified base directory, without generating a new JAR or ZIP file. This option enables the user to build up a customized JAR or ZIP file empirically, based on the requirements of their particular application. When this option is specified, -additionalFile, -additionalFilesDirectory, and -destination are ignored. The -extract option may be abbreviated to -x. By default, no extraction is done. The default base directory is the current directory.

► **-split [splitSize]**

Splits the source JAR or ZIP file into smaller JAR or ZIP files. No ZIP entries are added or excluded; the entries in the source JAR or ZIP file are simply distributed among the destination JAR or ZIP files. The split size is in units of kilobytes (1024 bytes) and specifies the maximum size for the destination files. The destination files are created in the current directory. They are named by appending integers to the source file name; any existing files by the same name are overwritten. For example, if the source JAR file is myfile.jar, the destination JAR files would be myfile0.jar, myfile1.jar, and so on. When this option is specified, all other options except -source and -verbose are ignored. The -split option may be abbreviated to -sp. The default split size is 2 MB (2048 KB).

► **-verbose**

Causes progress messages to be displayed. The -verbose option may be abbreviated to -v. The default is non-verbose.

► **-help**

Displays the help text. The -help option may be abbreviated to -h.

### 13.2.2 JarMaker example

In this example, the source JAR file is named myJar.jar and is in the current directory. To create a JAR file that contains only the classes mypackage.MyClass1 and mypackage.MyClass2, along with their dependencies, enter:

```
java utilities.JarMaker -source myJar.jar -requiredFile  
    mypackage/MyClass1.class,mypackage/MyClass2.class
```

Alternatively, the same function can be done with a Java program as shown here:

```
import utilities.JarMaker;  
// Set up the list of required files.  
Vector classList = new Vector();  
classList.addElement ("mypackage/MyClass1.class");  
classList.addElement ("mypackage/MyClass2.class");
```

```

JarMaker jm = new JarMaker ();
jm.setRequiredFiles (classList);
// Make a new jar file, that contains only MyClass1, MyClass2...
File sourceJar = new File ("myJar.jar");
File newJar = jm.makeJar (sourceJar); // smaller jar file

```

### 13.2.3 AS400ToolboxJarMaker

The AS400ToolboxJarMaker class is used to generate a smaller JAR or ZIP file from the shipped IBM Toolbox for Java JAR or ZIP file. In addition, you can use the AS400ToolboxJarMaker to:

- ▶ Extract desired files from a JAR or ZIP file
- ▶ Split a JAR or ZIP file into smaller JAR or ZIP files

You can use AS400ToolboxJarMaker in a program, or you can run AS400ToolboxJarMaker as a command line program, as shown here:

```
java utilities.AS400ToolboxJarMaker [ options ]
```

AS400ToolboxJarMaker extends the functionality of JarMaker by allowing the user to specify one or more IBM Toolbox for Java components, languages, or CCSIDs. You specify whether to include or exclude any JavaBean files that are associated with the specified components.

You must specify at least one of the following options:

- ▶ -requiredFile
- ▶ -additionalFile
- ▶ -package
- ▶ -extract
- ▶ -split
- ▶ -component
- ▶ -language
- ▶ -ccsid
- ▶ -ccsidExcluded
- ▶ -noProxy

If the following options are specified multiple times in a single command string, only the final specification applies:

- ▶ -source
- ▶ -destination
- ▶ -additionalFilesDirectory
- ▶ -extract
- ▶ -split
- ▶ -languageDirectory

Other options have a cumulative effect when you specify them multiple times in a single command string.

## Options

These options control what AS400ToolboxJarMaker will do:

► **-source sourceJarFile**

Specifies the source JAR or ZIP file from which to derive the destination JAR or ZIP file. The source file is typically jt400.jar. If a relative path is specified, the path is assumed to be relative to the current directory. If this option is specified as the first positional argument, the tag (-source) is optional. The -source option may be abbreviated to -s. The default is jt400.jar, in the current directory.

► **-destination destinationJarFile**

Specifies the destination JAR or ZIP file, which will contain the desired subset of the files in the source JAR or ZIP file. If a pathname is not specified, the file is created in the current directory. The -destination option may be abbreviated to -d. The default name is generated by appending "Small" to the source file name. For example, if the source file is jt400.jar, then the default destination file would be jt400Small.jar.

► **-requiredFile jarEntry1[,jarEntry2[...]]**

The files in the source JAR or ZIP file that are to be copied to the destination. Entries are separated by commas (no spaces). The specified files, along with all of their dependencies, will be considered required. Files are specified in the JAR entry name syntax, such as com/ibm/as400/access/DataQueue.class. The -requiredFile option may be abbreviated to -rf.

► **-additionalFile file1[,file2[...]]**

Specifies additional files (not included in the source jar) that are to be copied to the destination. Entries are separated by commas (no spaces). Files are specified by either their absolute path or their path relative to the current directory. The specified files will be included, regardless of the settings of other options. The -additionalFile option may be abbreviated to -af.

► **-additionalFilesDirectory baseDirectory**

Specifies the base directory for additional files. This should be the parent directory of the directory where the package path starts. For example, if the foo.class file in the com.ibm.mypackage package is located in the C:\dir1\subdir2\com\ibm\mypackage\ directory, then specify the base directory C:\dir1\subdir2. The -additionalFilesDirectory option may be abbreviated to -afd. The default is the current directory.

► **-package package1[,package2[...]]**

The packages that are required. Entries are separated by commas (no spaces). The -package option may be abbreviated to -p.

Package names are specified in standard syntax, such as com.ibm.component.

**Note:** The specified packages are simply included in the output. No additional dependence analysis is done on the files in a package, unless they are explicitly specified as required files.

► **-extract [baseDirectory]**

Extracts the desired entries of the source JAR into the specified base directory, without generating a new JAR or ZIP file. This option enables the user to build up a customized JAR or ZIP file empirically, based on the requirements of their particular application. When this option is specified, -additionalFile, -additionalFilesDirectory, and -destination are ignored. The -extract option may be abbreviated to -x. By default, no extraction is done. The default base directory is the current directory.

▶ **-split [splitSize]**

Splits the source JAR or ZIP file into smaller JAR or ZIP files. No ZIP entries are added or excluded. The entries in the source JAR or ZIP file are simply distributed among the destination JAR or ZIP files. The split size is in units of kilobytes (1024 bytes) and specifies the maximum size for the destination files. The destination files are created in the current directory. They are named by appending integers to the source file name, and any existing files by the same name are overwritten. For example, if the source JAR file is myfile.jar, then the destination JAR files would be myfile0.jar, myfile1.jar, and so on. When this option is specified, all other options, except -source and -verbose, are ignored. The -split option may be abbreviated to -sp. The default split size is 2 MB (2048 KB).

▶ **-component componentID1[,componentID2[...]]**

The IBM Toolbox for Java components that are required. Entries are separated by commas (no spaces) and are case insensitive. The -component option may be abbreviated to -c. See the list of components supported by the IBM Toolbox for Java.

▶ **-beans**

Causes inclusion of all JavaBeans files (classes, gifs) that are directly associated with the specified components. This option is valid only if -component is also specified. The -beans option may be abbreviated to -b. The default is no Beans.

▶ **-language language1[,language2[...]]**

Specifies the desired languages for the messages produced by the Toolbox classes. Entries are separated by commas (no spaces). The languages are identified by their Java locale name, such as fr\_CA (for Canadian French).

**Note:** The shipped jt400.jar file contains only English messages.

The -language option may be abbreviated to -l. By default, only English messages are included.

▶ **-languageDirectory baseDirectory**

Specifies the base directory for additional Toolbox language files. The path beneath this directory should correspond to the package name the language files. For example, if the desired MRI files are located in the /usr/myDir/com/ibm/as400/access/ and /usr/myDir/com/ibm/as400/vaccess/ directories, then the base directory should be set to /usr/myDir. The -languageDirectory option may be abbreviated to -ld. By default, language files are searched for, relative to the current directory.

▶ **-ccsid ccsid1[,ccsid2[...]]**

Specifies the CCSIDs which conversion tables should be included. Conversion tables for other CCSIDs will not be included. Entries are separated by commas (no spaces). The -ccsid option may be abbreviated to -cc. By default, all CCSIDs are included. See the list of CCSIDs and encodings that are specifically supported by the IBM Toolbox for Java.

▶ **-ccsidExcluded ccsid1[,ccsid2[...]]**

Specifies the CCSIDs whose conversion tables should not be included. Entries are separated by commas (no spaces). If a CCSID is specified on both the -ccsid and -ccsidExcluded, it is included, and a warning message is sent to System.err. The -ccsidExcluded option may be abbreviated to -cx or -ccx. By default, all CCSIDs are included. See the list of CCSIDs and encodings that are specifically supported by the IBM Toolbox for Java.

► **-noProxy**

Specifies that proxy-related class files should not be included. Proxy files are used internally by the Toolbox's Proxy Service. If the Proxy Service will not be used, the proxy files are not needed. The -noProxy option may be abbreviated to -np. By default, proxy files in the source JAR file are included.

► **-verbose**

Causes progress messages to be displayed. The -verbose option may be abbreviated to -v. The default is non-verbose.

► **-help**

Displays the help text. The -help option may be abbreviated to -h.

### 13.2.4 Example usage

To create a JAR file that contains only those files needed by the IBM Toolbox for Java CommandCall and ProgramCall classes, enter the following command:

```
java utilities.AS400ToolboxJarMaker -component  
CommandCall,ProgramCall
```

**Note:** You must be in the same directory as the jt400.jar file.

The resulting jt400Small.jar file is 928K (the jt400.jar file is 3217K). It includes all the supporting classes required. This drastically improves the amount of time required to download an applet that only uses these components.

## 13.3 Securing applications with SSL

Secure application communication with an iSeries server is important if your application is deployed on the Internet and the data is sensitive (such as banking information). SSL encrypts data at a socket layer before it is transmitted between a client and server. The concepts and environment configuration can be quite complex. However, the modifications required to an IBM Toolbox for Java program are very simple.

From OS/400 V4R4 on, there is the ability for host servers to communicate using SSL support. To provide this function in Java, the IBM Toolbox for Java now has a SecureAS400 object. SSL conversations can only take place between an IBM Toolbox for Java class and a V4R4 or later system that has SSL-enabled host servers.

Prior to discussing SSL, it is helpful to understand some elements of Internet security and how they relate to the iSeries server. The remainder of this chapter covers:

- An overview of the elements of transaction security available on the Internet
- A high-level description of the SSL protocol
- How to use Digital Certificate Manager (DCM) on iSeries to create an intranet certificate authority (CA) and server certificates
- How to apply the server certificate to the host servers
- How to receive a CA certificate into a Java class
- How to load the SSL classes into VisualAge for Java
- How to modify an existing application to use SSL
- How to verify that SSL is being used

### 13.3.1 Internet security elements

There is no one single answer to Internet security. Some people believe that by installing a firewall that their company network is safe.

Will a firewall alone shield your company from any inappropriate Internet access? No, security is not a matter of a single device or procedure. Security is a concept, a set of different security measures that are selected based on the needs of a specific installation.

Therefore, it is essential to first discuss the type of Internet security you need to achieve. Chances are that it is not just a firewall.

Figure 13-1 shows some of the elements that you can address when designing a total security solution for a company. However, this Redbook is not intended as an aid for deciding upon your company security policy. If you need help in forming a security policy for your company, it is better that you seek professional advice or services, such as the IBM SecureWay services.

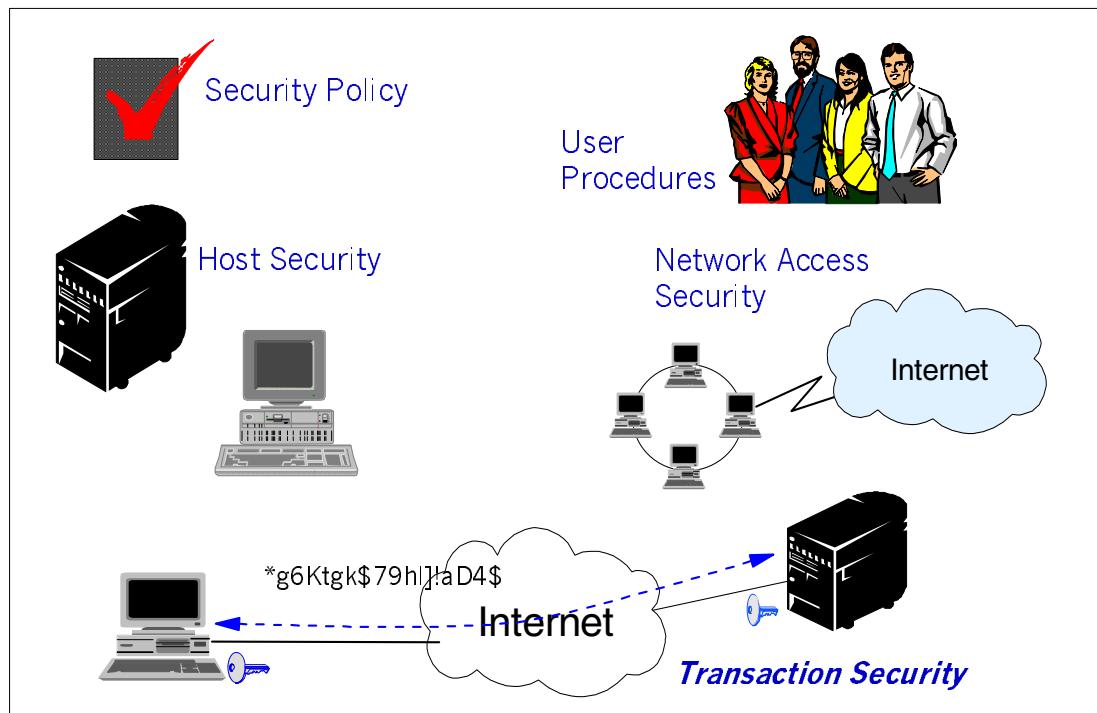


Figure 13-1 Internet security elements

First of all, a policy established by high-level management indicates how your company wants to deal with the Internet. It should define the level of security you want to achieve and how valuable or sensitive are the different types of information you possess. Various Internet security features, such as cryptography or host system security functions, can help you to implement what you design.

In addition, users need to be educated to follow and maintain the implemented security procedures, as well as to observe specific rules when acting as Internet clients.

### 13.3.2 Transaction security and Secure Sockets Layer

Transaction security includes several basic elements, such as:

- ▶ Confidentiality/privacy
- ▶ Integrity
- ▶ Authentication
- ▶ Accountability

SSL is the Secure Sockets Layer protocol defined by Netscape Communications Corporation. It provides a private channel between a client and server that ensures privacy of data, authentication of session partners and message integrity.

*Digital certificates* are used for session partner authentication. Server authentication is common. Client authentication is not yet common, but it is growing in popularity.

Keys are the base for end-to-end information encryption. Figure 13-2 provides a high-level view of SSL and transaction security.

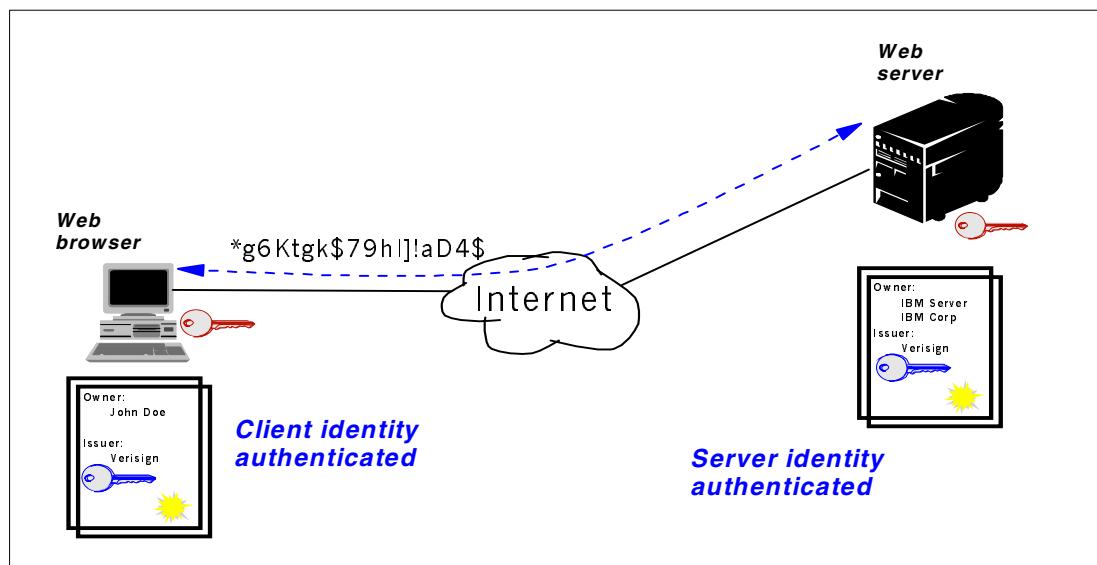


Figure 13-2 Transaction security

You must re-write TCP/IP applications to use SSL. Primarily, SSL is used by HTTP (HTTPS) for Web browsing. In OS/400 V4R3, Directory Services Server (LDAP) is SSL enabled. With OS/400 V5R1, many applications including the host servers have been SSL enabled by the licensed products as shown in Table 13-1.

Table 13-1 iSeries SSL Licensed Program Products

Installed Cryptographic Access Provider product	Required Client Encryption product	Key length
5722-AC2	5722-CE2	56 bit
5722-AC3	5722-CE3	128 bit

When you order OS/400, you are shipped the appropriate version of the 5722-ACx and 5722-CEx products in accordance with US export laws.

## Confidentiality

When a packet travels across a standard network, it is possible to use a *packet sniffer* to passively read the message. This means packets travelling a network can be read without either the sender or receiver ever knowing. To overcome this, messages should be encrypted to assure confidentiality.

Confidentiality means that the contents of the messages remain private as they pass through the Internet. Without confidentiality, your computer broadcasts the message to the network, similarly to shouting the information across a crowded room. *Encryption* ensures confidentiality.

## Integrity

For example, you may want to know if the data received is the same as the data that was sent. You can determine this through two possible solutions: *digital signature (or hashing)* and *encryption*.

The sending system calculates a hash value based on the message being sent. The hash value is appended to the transmission. The receiving system uses the same calculation to generate a value. The receiving system then compares the calculated value with the received value. If the values are different, then it assumes that the data changed. To provide more bullet proof security, the message should first be encrypted using an appropriate encryption algorithm.

Integrity means that the messages are not altered while being transmitted. If a router or other network device inserts, deletes, or garbles the message as it passes by, the receiver would detect the modification. Without integrity, you have no guarantee that the message you sent matches the message that was received. Encryption and digital signature ensure integrity.

## Authenticity

Consider the scenario where you want to know who is at the other end of a Web site to test its authenticity. One way to find out is through the use of digital certificates and digital signatures (see Figure 13-3).

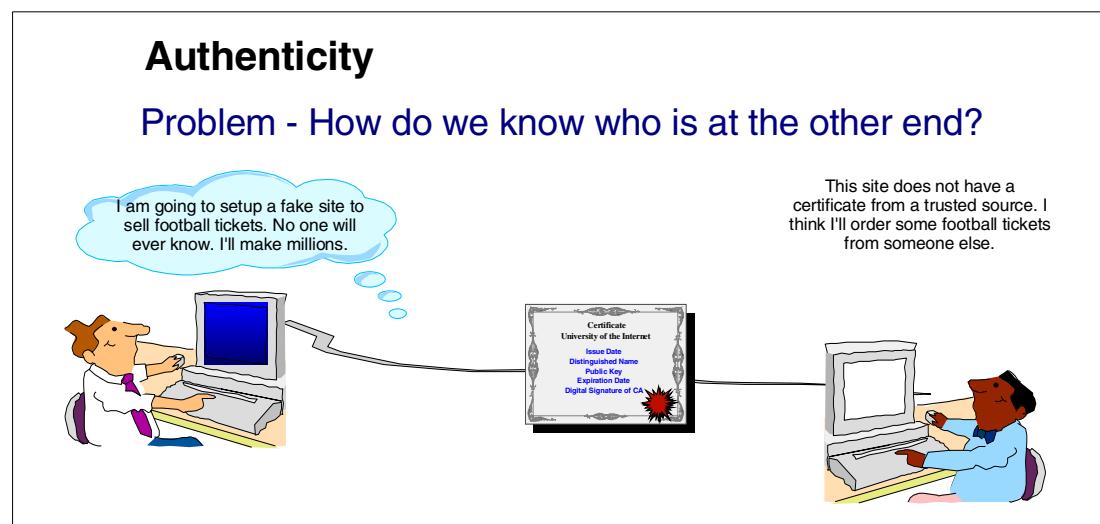


Figure 13-3 Verifying identity: Digital certificates and digital signatures

Authenticity means that you know who you are talking to and that you trust that person. Without authenticity, you have no way to be sure that anyone is who they say they are. *Authentication* through digital certificates and digital signatures ensure authenticity.

There are two ways in which the server uses authentication:

- ▶ Digital signature
- ▶ Digital certificates

Digital signature ensures accountability. But how do you know if the person sending you a message is who they say they are?

To ensure accountability look at the sender's *digital certificate*. A public key certificate is issued by a trusted third party known as the *certifying authority (CA)*. The browser and server exchange information, including their public key certificate. SSL uses the information to identify and authenticate the sender of the certificate.

You can think of a digital certificate as being like a credit card with your picture on it and a picture of the bank president with his arm around you. A merchant will trust you more because not only do you look like the picture on the credit card, but the bank president trusts you, too.

You base your trust for the authenticity of the sender on whether you trust the third party (a person or agency) that certified the sender. The third party or certification authority issues digital certificates.

How can you ensure that the person sending the message is really trustworthy? Let us use an example to illustrate this point.

If you wake up one day feeling ill, you may decide to visit a doctor. You can select a doctor from your phone book and go to their office for a visit. Once you get to the office, how can you be sure that the person about to examine you is really a doctor? After all, you have never met this person before. They may look like a doctor and act like a doctor, but who's to say that this person has successfully completed all of the training necessary to become a doctor?

You need certification by a trusted third party to reassure you that this person really is a doctor. The doctor probably has a diploma on the wall stating that they have successfully completed their training. If the diploma is from a well-known school, you would probably be reassured that you are about to be examined by a real doctor. But what if the diploma is from the medical school of a correspondence school whose name you do not recognize? You may not be so reassured.

Authentication works the same way. Trusted third parties verify that the server really is who it claims to be. This verification is provided with a digital certificate – the digital equivalent of your doctor's diploma hanging on the wall. You base your trust for the authenticity of the server on whether you trust the third party that certified the server – the school that issued the diploma. That third party is called a *certifying authority (CA)*.

The term *trusted root* is given to a trusted certifying authority (CA) on your server. A *trusted root key* is the key that belongs to the CA.

You can use authentication server to client (server authentication) or client to server (client authentication). Server authentication is described above. The clients authenticate the servers. With client authentication, the client is authenticated by the server. For example, you may use client authentication if a server contained hospital patient information, to verify that the client attempting to access the data is really who they said they are before allowing them access to patient records.

## **Accountability**

If you need to prove that specific transactions took place, you need to implement some form of accountability.

To prove that a transaction occurred, combine all the previous techniques. First, calculate the hash code of the data to assure data integrity. The data is then encrypted using the keys derived from the public key exchange. This assures the identity of the session partners. This is used in combination with a time stamp in the data to provide a log of the transactions.

Accountability means that both the sender and receiver agree that the exchange took place. Without accountability, the target application user can easily deny that the data arrived. You can use digital signatures to ensure accountability. However, accountability *is not* part of the SSL protocol since it requires an application to perform the tasks described.

## 13.4 Digital certificates and certificate authority

A digital certificate identifies a user or a system and is required before you can use SSL. Once a server has a digital certificate, SSL-enabled browsers, such as the Netscape Navigator, can communicate securely with the server using SSL. A digital certificate consists of:

- ▶ Owner's distinguished name
- ▶ Owner's public key
- ▶ Digital signature of certificate authority
- ▶ Name of the CA
- ▶ Issue date of certificate
- ▶ Certificate expiration date
- ▶ Serial number

Plus, digital certificates have the following characteristics:

- ▶ Digital certificates are digital documents that validate the identity of the certificate's owner.
- ▶ There are three types of digital certificates: CA, server, and client certificates.
- ▶ Digital certificates contain a public key, which binds it to an identity.
- ▶ Digital certificates are created by trusted third parties called certificate authorities (CA).
- ▶ Digital certificates can be distributed freely.
- ▶ A digital signature in the digital certificate prevents tampering.

A certificate authority issues a digital certificate. CAs are entities that are trusted to properly issue certificates and have controls in place to prevent fraudulent use. They are the equivalent to the Department of Motor Vehicles for a driver's license. An individual may have many certificates from different CAs just like we may have many forms of identification (passport, credit card, gym membership card, and so on). If you trust a CA, you can be reasonably assured that any certificate they issue properly represents the individual that is holding it. The CA charges a fee for issuing a certificate.

- ▶ CAs broadcast their public key and distinguished name.
- ▶ People add them as trusted root keys to Web servers and browsers.
- ▶ Your server trusts anyone who has a certificate from that CA.
- ▶ There are several common CAs in the marketplace.
- ▶ Servers and browsers are shipped with several default trusted root keys and more can be added as needed.

Some examples of universally recognized Internet CAs include:

- ▶ Thawte Consulting
- ▶ VeriSign, Inc.
- ▶ IBM World Registry

For testing purposes or for applications that will be used exclusively in an intranet environment, you may issue digital certificates using an intranet certificate authority. The iSeries server with Digital Certificate Manager can act as an intranet certificate authority. However, be aware that the real cost of being your own CA can escalate. It is often cheaper to purchase a valid certificate from a well-known certificate authority.

For secure communications, the receiver must trust the CA that issued the certificate, whether the receiver is a browser or a server. Anytime a sender signs a message, the receiver must have the corresponding CA certificate and public key designated as a trusted root key.

## 13.5 iSeries implementation of Digital Certificate Management

You can configure your iSeries server as an intranet certificate authority. Digital Certificate Manager is a Web browser-based administration facility that allows you to create, manage, and use certificates within an enterprise and with partners of an enterprise. You can use DCM to request digital certificates from such Internet CA as VeriSign and Thawte.

DCM allows you to create your own intranet CA. You can then use the CA to dynamically issue digital certificates to servers and users (client certificates) on your intranet. When you create a server certificate, DCM automatically generates the private key and public key for the certificate. You can also use DCM to register and use digital certificates from Verisign or other commercial organizations on your intranet or the Internet.

You should install the following products in the iSeries server before implementing SSL connections:

- ▶ IBM HTTP Server for iSeries (5722-DG1) licensed program
- ▶ Base operating system option 34 (Digital Certificate Manager)
- ▶ The IBM Cryptographic Access Provider for iSeries licensed program (5722-AC2 or 5722-AC3) to provide server-side encryption
  - 5722-AC2 provides 56-bit encryption
  - 5722-AC3 provides 128-bit encryption
- ▶ The IBM iSeries Client Encryption licensed program (5722-CE2 or 5722-CE3). Client Encryption provides the Java classes and utilities used by the IBM Toolbox for Java classes on the client side.

The cryptographic products above determine the maximum key length permitted for cryptographic algorithms on your iSeries server. Government export or import regulations determine which version is available in your country.

**Note:** To use all the options available in DCM, you must have \*SECOFR and \*SECADM authority.

To access DCM, open a browser session (see the example in Figure 13-4 on page 514).

### 13.5.1 Changing the authority of the directory

To help you meet the SSL legal responsibilities required when using cryptography algorithms, the directory that contains the files is shipped with public authority \*EXCLUDE. You must change the authority of the directory to allow access by only those users authorized to use encryption algorithms.

Use OS/400 object security to control access to the client encryption files by completing the following steps:

1. On your server, enter the following command:  
`wrklnk '/QIBM/ProdData/HTTP/Public/jt400/*'`
2. Select option 9 in the SSL56 or SSL128 directory.
3. Ensure that \*PUBLIC has \*EXCLUDE authority.
4. Give \*RX authority to the directory to individual or groups of users who need access to the SSL files.

**Note:** You cannot deny access to the SSL files to users that have \*ALLOBJ special authority.

### 13.5.2 Configuring a digital certificate environment

You can use your iSeries server to configure a digital certificate environment. Once the environment is configured, you can use certificates to enable SSL communication.

Perform the following series of steps to configure an intranet digital certificate environment using the iSeries server as a CA:

1. Use DCM to create an intranet CA in one or more iSeries servers.
2. Using DCM, use the intranet CA to issue server certificates, which can be used in the local server (same iSeries server where the CA is configured) or exported to a remote server.
3. For the clients to recognize and trust the server certificates issued by the intranet CA, the client must download and designate the CA certificate as a trusted root.

## 13.6 Using a self-signed certificate for SSL

This section explains how to create a self-signed certificate using your iSeries server as an intranet CA. Because self-signed certificates are not recognized by client applications as coming from a trusted third party, you should not use them in customer transaction situations over the Internet. Use them only on your test and development systems and for demonstration purposes. You can also use a self-signed certificate for intranet applications.

To obtain a self-signed certificate, complete the following steps:

1. Create an intranet CA.
2. Create a server certificate with your intranet CA.
3. Configure your servers to use the server certificate.

### 13.6.1 Creating an intranet certificate authority

DCM allows you to create your own intranet CA in your iSeries server and use it to issue server and client certificates for testing purposes or applications within your organization. This section outlines the steps that you must perform to create a CA on your iSeries server. You only need to perform these steps if the system administrator did not previously create an intranet CA and if you want to use your iSeries server to issue intranet server certificates.

To create an intranet CA in your iSeries server, follow these steps:

1. Start the HTTP \*ADMIN server on your iSeries server. From the command line, enter this command:  
`STRTCPVR SERVER(*HTTP) HTTPSVR(*ADMIN)`

2. Access the AS/400 Tasks page from your browser by entering this URL:  
`http://System_name:2001`
3. You are prompted to enter your user name and password. Sign on with a user that has \*SECOFR and \*SECADM authority. The AS/400 Tasks page is displayed as shown in Figure 13-4.



Figure 13-4 AS/400 Tasks page

4. Click **Digital Certificate Manager**.

5. Click **Create a Certificate Authority (CA)**.

**Note:** If a Certificate Authority (CA) was previously created on your system, the Create a Certificate Authority link is not displayed.

6. Complete the Create a Certificate Authority form as shown in Figure 13-5. Replace the field values as appropriate with your organization information.

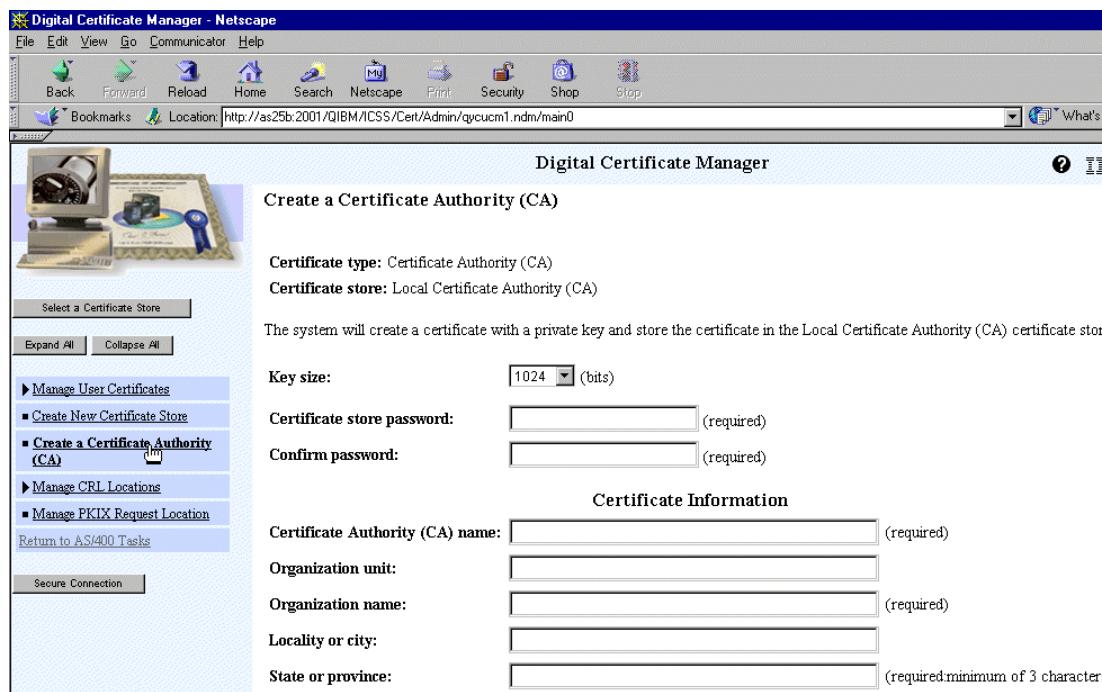


Figure 13-5 Creating an intranet certificate authority

7. Click **Continue**.
8. After DCM processes the form, it stores a copy of the CA certificate in the CA default keyring file.

DCM displays the page shown in Figure 13-6.

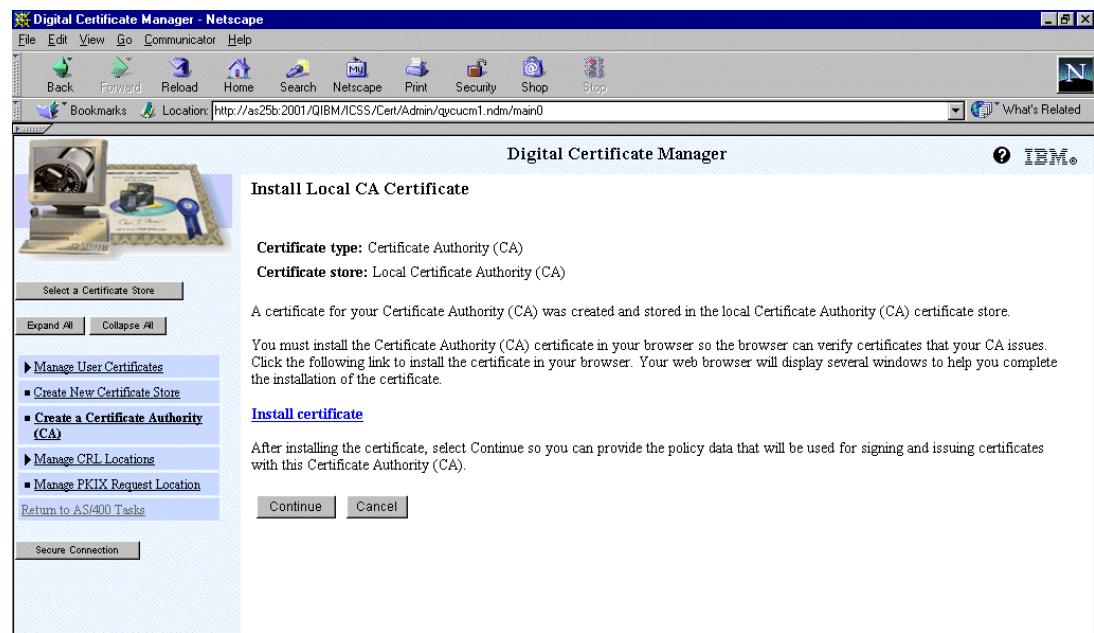


Figure 13-6 CA certificate was created successfully

9. Click **Continue**.

Complete the CA Policy Data form to set the certificate policy for your CA (see Figure 13-7). This is where you define whether your CA can issue and sign user certificates. If the CA can issue user certificates, indicate the length of time for which the certificates will be valid.

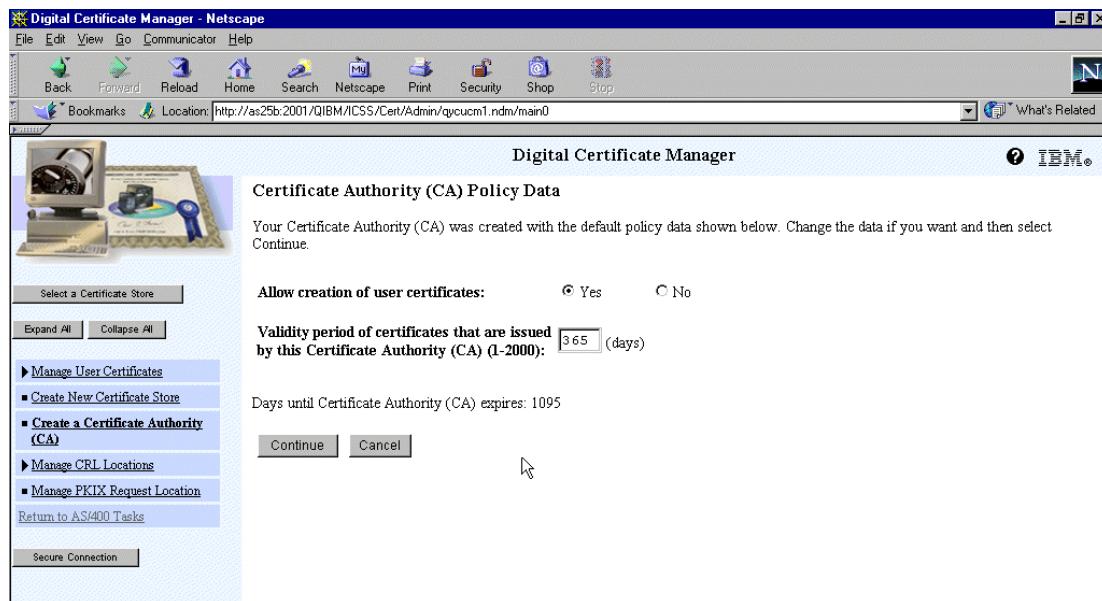


Figure 13-7 Certificate Authority Policy

**10. Click Continue.**

11. The next display that appears shows the message that the policy data for the Certification Authority(CA) was accepted. You are invited to create a default server certificate store(\*SYSTEM) and a server certificate by your certificate authority. This allows server authentication by users that use this system as a server. See Figure 13-8.

**12. Click Continue.**

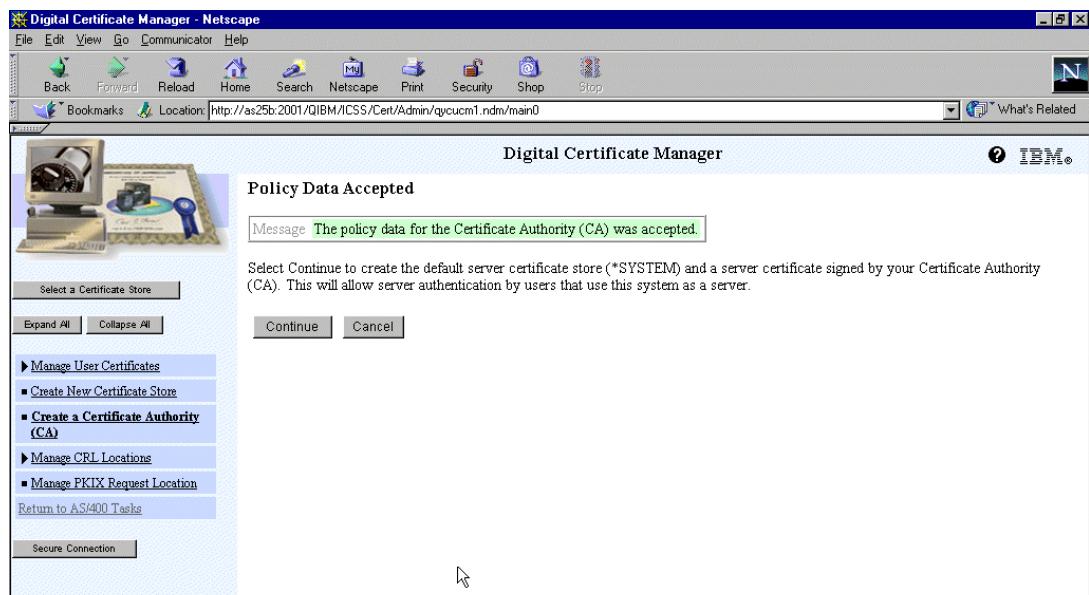


Figure 13-8 Invitation to create a default server certificate store(\*SYSTEM)

## 13.6.2 Creating a server certificate with your intranet CA

Immediately after creating the intranet CA, DCM leads you to create a default certificate store(\*SYSTEM) and a server certificate. To use SSL, your server must have a digital certificate. When you create a server certificate in DCM, the server certificate and keys are stored in the default /QIBM/USERDATA/ICSS/CERT/SERVER/DEFAULT.KDB directory and file. Make sure to back it up.

**Note:** When you create a server certificate, DCM stores a copy of the CA certificate in the server key ring and designates it as a trusted root.

Complete the following steps:

1. Complete the Create a System Certificate form, as shown in Figure 13-9, by replacing the field values with your organization information.

The options for the key size are determined by the IBM Cryptographic Access Provider (5722-ACx) Licensed Program Product installed in your system. This is the key size that is used to generate your public and private keys. The higher the value is, the more secure the conversation is.

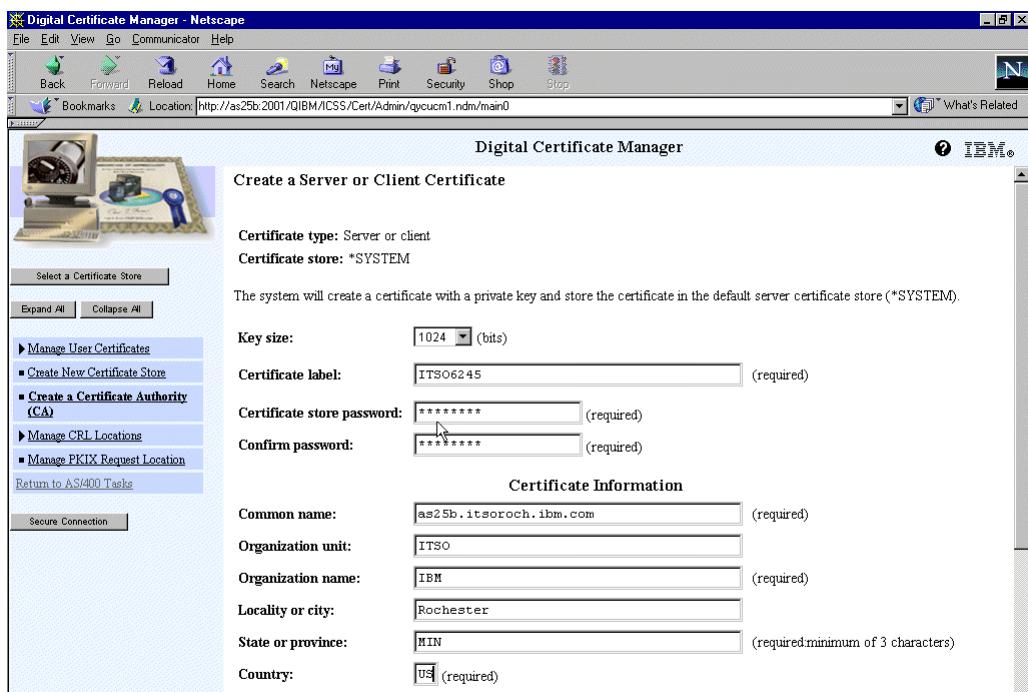


Figure 13-9 Create a Server or Client Certificate page

You can specify the fully qualified name of the iSeries server into the common name field. This is the name used to describe your server. You can enter any common name, although the fully qualified TCP/IP host name is usually used for the common name.

2. Click **Continue**.
3. The Server Certificate Created Successfully page displays the message Your certificate was created and placed in the \*SYSTEM certificate store. You are then asked to select which applications will use this certification. Scroll down the page and select **OS/400 - Host Servers**. See Figure 13-10 on page 518.

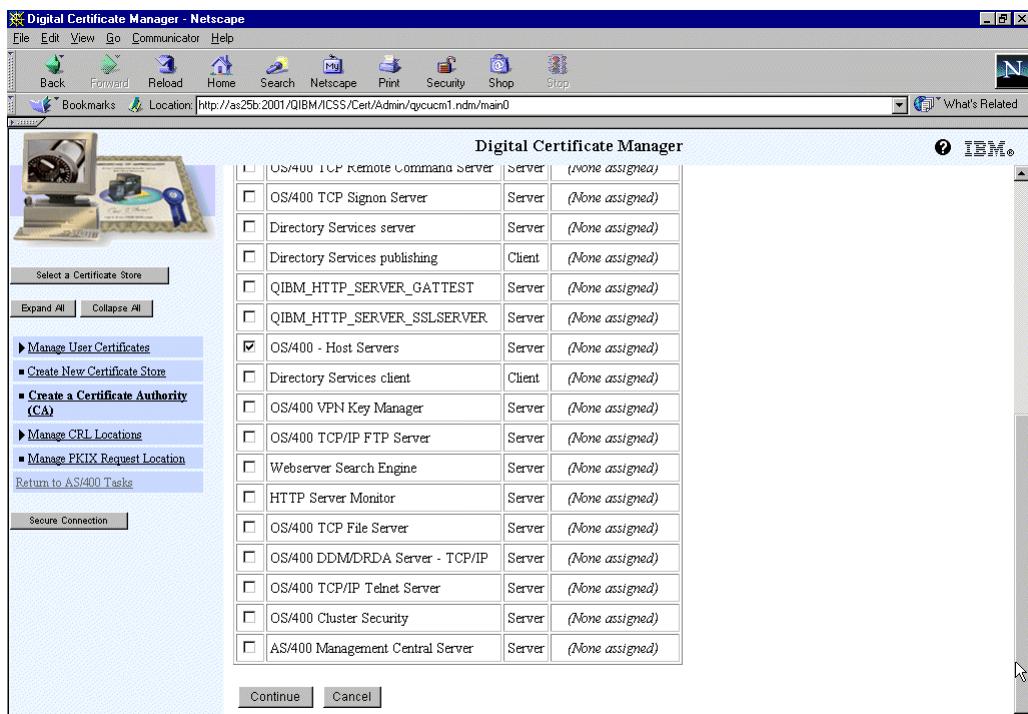


Figure 13-10 Server Certificate Created Successfully page

4. Click **Continue**.
5. The Application Status page displays the message The application you select will use this certificate.
6. Click **Cancel**. You don't need to create \*OBJECTSIGNING at this time.
7. End and restart host servers by issuing the following OS/400 commands:

```
ENDHOSTSVR *ALL
STRHOSTSVR *ALL
```

## 13.7 Using a server certificate from an Internet CA

To conduct commercial business on the Internet, you should request your server certificate from an Internet certificate authority, such as VeriSign or Thawte, who are widely known by clients browsers and servers.

For your private Web network within your own company, university, or group, or for testing purposes, you can act as your own CA by using Digital Certificate Manager. Section 13.6, "Using a self-signed certificate for SSL" on page 513, explains this procedure.

This section explains how to obtain a server certificate from an Internet certificate authority.

To use a server certificate issued by an Internet CA, you must complete these steps:

1. Request the server certificate from an Internet CA.
2. Receive a server certificate for this server.
3. Configure host servers to use this certificate.

To use SSL for secure Web serving, your server must have a digital certificate. You can use an intranet CA to issue a server certificate (see 13.6, "Using a self-signed certificate for SSL" on page 513). Or, you can use an Internet CA.

When you choose to use an Internet CA to issue a server certificate, you must first request the certificate. To request a certificate, use the following steps:

1. From the DCM page, click the **Select a Certificate Store** button in the left-hand frame. Select **\*SYSTEM**, and click **Continue**.
2. Input the password for the **\*SYSTEM** store, and click **Continue**.
3. Click **Create Certificate**, and select the **Server or client certificate** radio button. Click **Continue**.
4. For the type of Certificate Authority(CA), select **VeriSign or other Internet Certificate Authority** as shown in Figure 13-11.

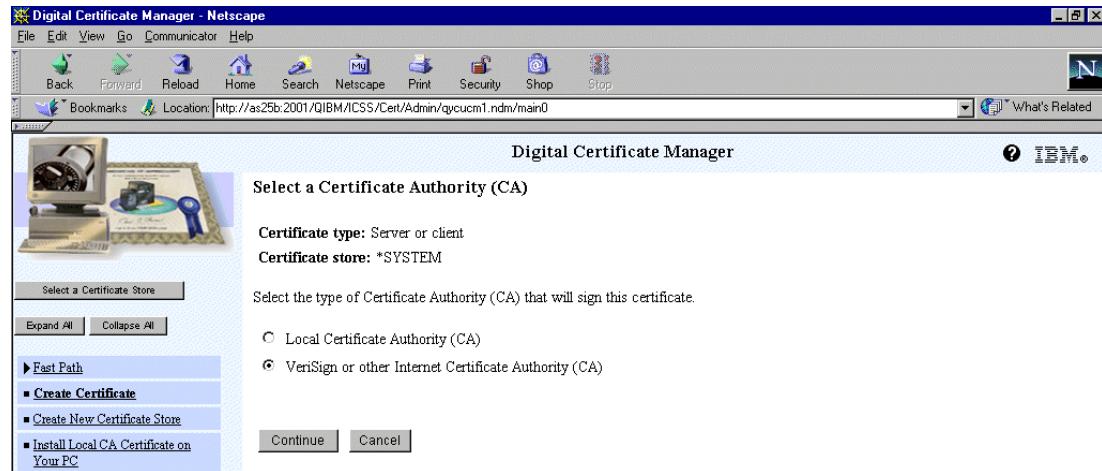


Figure 13-11 Requesting a certificate from VeriSign or other Internet certificate authority

5. Click **Continue** to display the Create Certificate form.
6. Complete the Create Certificate form as show in Figure 13-12 on page 520.

The options for the key size are determined by the IBM Cryptographic Access Provider (5722-ACx) licensed program installed in your system. This is the key size that generates your public and private keys.

Figure 13-12 Requesting a server certificate from an Internet CA

You can specify the fully-qualified name of the iSeries server in the system name field. This is the name used to describe your server. You can give the server any name, although the fully qualified TCP/IP host name is usually used for the server name.

#### 7. Click **Continue** to process the Create a Certificate Request form.

You will receive the Server Certificate Request Created page as shown in Figure 13-13.

```

-----BEGIN NEW CERTIFICATE REQUEST-----
MIIBrTCARYCAQAwBtELMAkGA1UEBhMCVVMxDDAKBgNVBAgTAO1JTjESMBAGA1UE
BxMjUm9jaGVzdGvycQwwcYDVQQKEwNJQk0xDTALBgNVBAstBE1UU08xHzAdBgNV
BAMTFk1ZU11TVEVNLk1ZQD9NUEFOW5SDT00w28wQVJKoZIhvCNQERBBQADgYOA
MIGjAqGBALIdcPdmqnsHyUfInb3KD94gONxxVEcJuGfOfbVG+ehT2oheHAbcMRySO
ZQxdSsQSFw4Ncjv1BHea6nYjYC1LqUPy7odbhEh/jLgNKN584DjEGa3VO10Tb2fJ
dqwkLxF+HQNeXgXdeRThpYF8pItw5PYAzvObP/dvBtX5Z23ZgPAgMBAcGADAN
BgkqhkiG9wOBQQAFAObgCpzndjsnpS4u8/un/s0/wxtjeW9gLnbSmCt520wlaAk
v7jhpxyt6jqy3e0gvA4yikmGKQYhRgY7ORW41+suP3f2bzAnE8Yh+qSwaiDuvH
EUd8W2GyND2N+WHtr7KLSY/m94DItinIH9Ef+7mlkL6el1xIEJjdjcrRCTBee/ON
Qg==
-----END NEW CERTIFICATE REQUEST-----

```

Figure 13-13 Server certificate request generated by DCM

**Note:** Do not click Done or close the browser yet. You need to cut and paste the certificate request when you submit the Certificate Signing Request to the Internet CA.

8. Copy the Server Certificate Request to your clipboard. Start at -----BEGIN NEW CERTIFICATE REQUEST----- and end at -----END NEW CERTIFICATE REQUEST----- . Click **Done** to close the page.
9. Follow your Internet CA procedures to paste the certificate request. For example, to request a certificate from VeriSign, follow the instructions that are described on the Web site: <http://www.verisign.com>

When the Internet CA is satisfied that you meet all of its requirements, it will e-mail the secure server certificate to you. You should receive it in three to five business days. Other certificates authorities have their own procedures.

### 13.7.1 Receiving a server certificate for this server

After you receive the certificate from the Internet CA, you need to copy the signed server certificate to a text file that DCM can access when you perform the Receive server certificate task. Perform these steps:

1. Copy the signed server certificate presented to you by the Internet CA to your clipboard. Include the -----BEGIN CERTIFICATE REQUEST----- and -----END CERTIFICATE REQUEST----- sections of the certificate
2. Paste the signed server certificate in your clipboard into an empty .txt file.
3. Save the file in your iSeries server integrated file system. Use a mapped network drive and save the .txt file that contains the server certificate issued by the Internet CA in the following path (you can actually store this file where ever you want):  
 /QIBM/USERDATA/ICSS/CERT/SERVER/rvcert.txt
4. Back to the left-hand side of the DCM, select **Manage Certificates-> Import certificate**. In the Import Certificate page, select **Server or client**. Click **Continue**.
5. Fill in the fully qualified path and file name. Then click **Continue**.

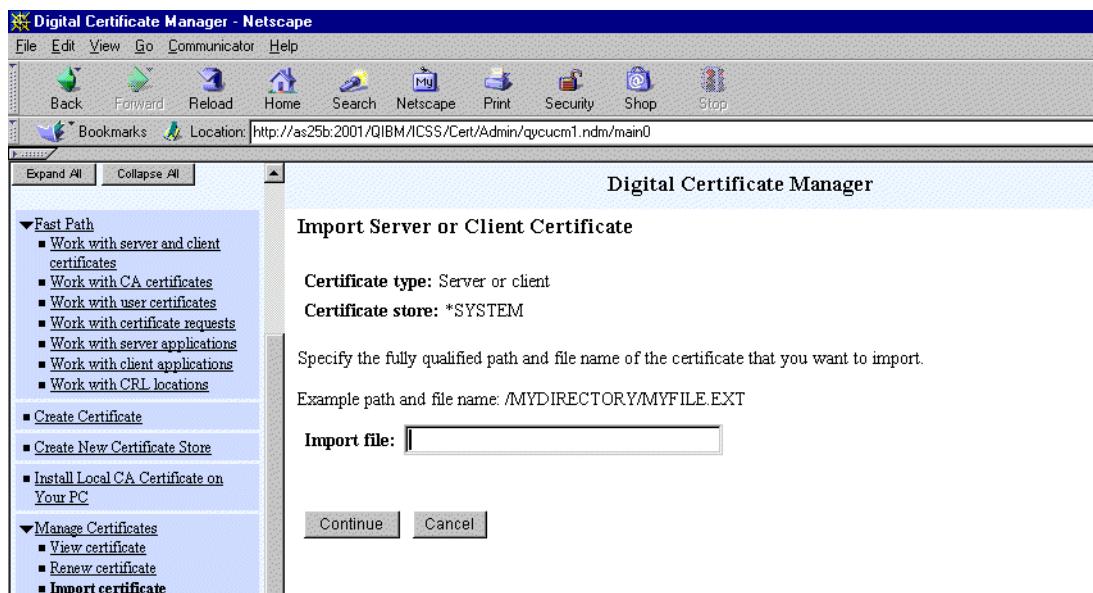


Figure 13-14 Receiving a server certificate issued by an Internet CA

6. The Certificate Received page is displayed. You should now apply the certificate to the iSeries servers. Select **OS/400 - Host Servers** to enable SSL for the host servers.

## 13.8 Using a certificate with the IBM Toolbox for Java

Set up the client (the IBM Toolbox for Java classes) to exchange encrypted data. The procedure for this step depends on the kind of certificate you used when you set up SSL on your server:

- ▶ You set up the client when using a server certificate from a trusted authority.
- ▶ You set up the client when using a self-signed certificate.

### 13.8.1 Using a certificate from a trusted authority

The IBM Toolbox for Java ships a keyring file that supports server certificates from a set of trusted authorities, represented by the following companies:

- ▶ IBM World Registry
- ▶ Integriion Financial Network
- ▶ RSA Data Security, Inc.
- ▶ Thawte Consulting
- ▶ VeriSign, Inc.

The keyring file already supports certificates that you get from one of these trusted authorities. All you need to do is obtain the ZIP files that contain the encryption algorithms and add it to your CLASSPATH statement.

To use the certificate, complete the following steps:

1. Select the directory where you want to put the ZIP files.
2. Download the version of SSL that you want to use by copying the ZIP files into the selected directory:
  - For 56-bit encryption (used with licensed program 5722-CE2), copy **/QIBM/ProdData/HTTP/Public/jt400/SSL56/sslightx.zip**
  - For 128-bit encryption (used with licensed program 5722-CE3), copy **/QIBM/ProdData/HTTP/Public/jt400/SSL128/sslightu.zip**
3. Add the ZIP file to your CLASSPATH statement.

### 13.8.2 Using a self-signed certificate

When you choose not to use a certificate from a trusted authority, you must download the server certificate (to each server that has a self-signed certificate) so that the IBM Toolbox for Java classes can use it. You also have to get the ZIP files that contain the encryption algorithms and add them to your CLASSPATH statement.

To use the self-signed certificate, complete the following steps (see Figure 13-15 on page 524):

1. Select the directory where you want to put the ZIP files.
2. Download the version of SSL that you want to use by copying both the encryption algorithms and the utilities you need to work with a self-signed certificate:
  - For 56-bit encryption (used with the licensed programs 5722-CE2) copy **/QIBM/ProdData/HTTP/Public/jt400/SSL56/sslightx.zip**, **cwk.zip**, and **ssltools.jar**.
  - For 128-bit encryption (used with the licensed programs 5722-CE3) copy **/QIBM/ProdData/HTTP/Public/jt400/SSL128/sslightu.zip**, **cwk.zip**, and **ssltools.jar**.
3. Add **ssltools.jar** and the ZIP files to your CLASSPATH statement.

4. Create a directory on your client named <SSL>\com\ibm\as400\access, where <SSL> is the directory where you copied the JAR and ZIP files.
5. From a command prompt within the <SSL> directory on your client, run the command:  

```
java utilities.KeyringDB com.ibm.as400.access.KeyRing -connect <systemname>:<port>
```

Here, <port> is the server port of any of the host servers. For example, you can use 9476, which is the default port for the secure sign-on server on the iSeries server.  
**Note:** You must use com.ibm.as400.access.KeyRing because it is the only location where the IBM Toolbox for Java looks for your certificates.
6. Type the number of the certificate authority certificate that you want to add to your server. Be sure to add the CA certificate and not the site certificate.
7. When you are prompted to enter a certificate name, you can type any alphanumeric string.  
**Note:** You need to run KeyringDB to each server that has a self-signed certificate to add each certificate to the KeyRing class. On each iSeries on which you want to use SSL connections, run the following command to add the certificates:  

```
java utilities.KeyringDB com.ibm.as400.access.KeyRing connect <systemname>:<port>
```

```

Microsoft(R) Windows NT DOS
(C)Copyright Microsoft Corp 1990-1996.

C:>cd ssl

C:\SSL>md com\ibm\as400\access

C:\SSL>set CLASSPATH=%CLASSPATH%;c:\ssl\sslttools.jar;c:\ssl\sslightu.zip;c:\ssl\cfwk.zip

C:\SSL>java utilities.KeyringDB com.ibm.as400.access.KeyRing -connect MyServer:9
476
AS/400 Toolbox for Java

(C) Copyright IBM Corporation 2000. All rights reserved.
U.S. Government users Restricted Rights - Use, duplication or
disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
Licensed Materials - Property of IBM

pkgout : com.ibm.as400.access

Connecting to MyServer:9476
CA Certificate - Number 0

      Key : RSA/1024 bits
      Subject: s10a8858, IBM, US
      Issuer: T3 Education, Rochester, Friday test, IBM, us
      Valid from: Tue Mar 27 15:17:57 CST 2001
      Valid to: Thu Mar 28 15:17:57 CST 2002
      13:85:3B:6B:A6:59:04:05:B6:E4:4B:2E:94:27:C7:FA

Site Certificate - Number 1

      Key : RSA/1024 bits
      Subject: T3 Education, Rochester, Friday test, IBM, us
      Issuer: T3 Education, Rochester, Friday test, IBM, us
      Valid from: Thu Mar 22 14:17:10 CST 2001
      Valid to: Mon Mar 22 14:17:10 CST 2004
      B6:55:E1:5E:AC:1A:30:05:D9:F7:40:B8:F7:BD:47:29

Select certificate [default = 0, 99 to cancel]: 0
Name for chosen certificate:MyCertificate
**   Successful   **

C:\SSL>

```

*Figure 13-15 Adding the CA certificate to the KeyRing.class file*

Once you complete these steps, you are finished setting up the self-certificates. You can run the application using an SSL connection.

## 13.9 Modifying an application to use SSL with VisualAge 3.5

After you download the cfwk.zip and the sslichtx.zip or sslightu.zip archive files, you need to import them into the VisualAge for Java environment. If it was necessary to create your own com.ibm.as400.access.KeyRing class, you also need to import it into VisualAge for Java before you can run any modified applications.

Once the files are imported into VisualAge for Java, modifying a IBM Toolbox for Java program to use SSL is very simple.

### 13.9.1 Importing the required classes

To import the required classes, follow these steps:

1. Start VisualAge for Java.
2. Select the **IBM Enterprise Toolkit for AS400** project. If you are using a different edition of VisualAge, select the project containing the IBM Toolbox for Java classes.
3. Create an Open Edition of this project.
4. Import the Java archive by selecting **File-> Import**. Select the **Import from a Jar file** option, and click **Finish**.
5. Use the Browse button to locate the ssllightx.zip or ssllightu.zip archive you previously downloaded. This is illustrated in Figure 13-16.

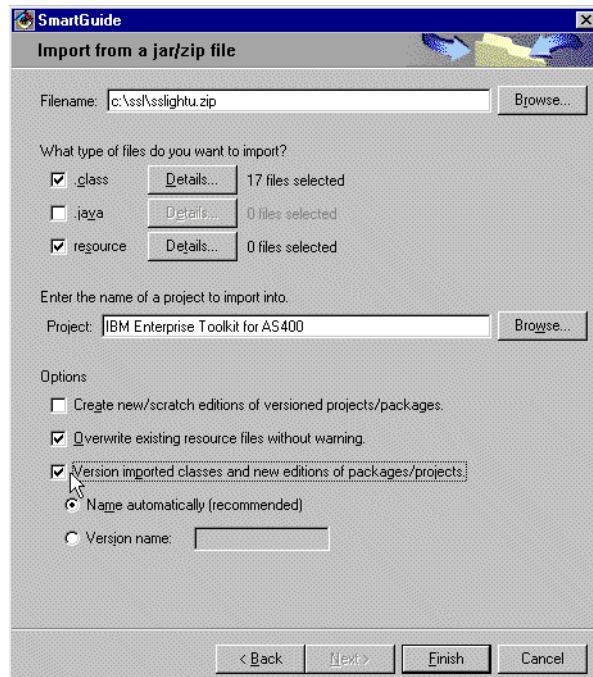


Figure 13-16 Importing SSL support into VisualAge for Java

6. You also need to import the cfwk.zip file using the above procedure.

If you used the SSLTools.zip archive to create a customized KeyRing class, you need to perform the following additional steps:

1. Locate the existing **com.ibm.as400.access.KeyRing** class, and delete it.
2. Import the new **com.ibm.as400.access.KeyRing** class by using the Import from a directory option. This is illustrated in Figure 13-17 on page 526.

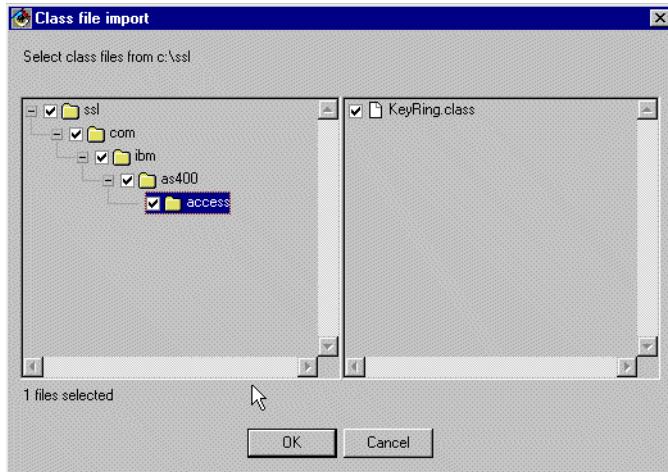


Figure 13-17 Importing a modified KeyRing class

You have now completed all of the steps that are necessary to develop SSL-enabled IBM Toolbox for Java applications to communicate with iSeries servers that run OS/400 V4R4 or later. Now, it is time to modify an application to use SSL.

### 13.9.2 Modifying the program

For the IBM Toolbox for Java classes to use SSL to communicate with the iSeries server, it is only necessary to change the AS400 objects to SecureAS400 objects. If you are using JDBC, set the secure property to "true". This causes JDBC to use a SecureAS400 object.

In VisualAge for Java, you can use the Morph into feature to change an AS400 object to a SecureAS400 object. The following steps convert the TBVisual.RLFPEExample program to use SSL:

1. Open the **TBVisual.RLFPEExample** class in the Visual Composition Editor (VCE).
2. Right-click the **AS400** object, and select the **Morph into** option, as illustrated in Figure 13-18.

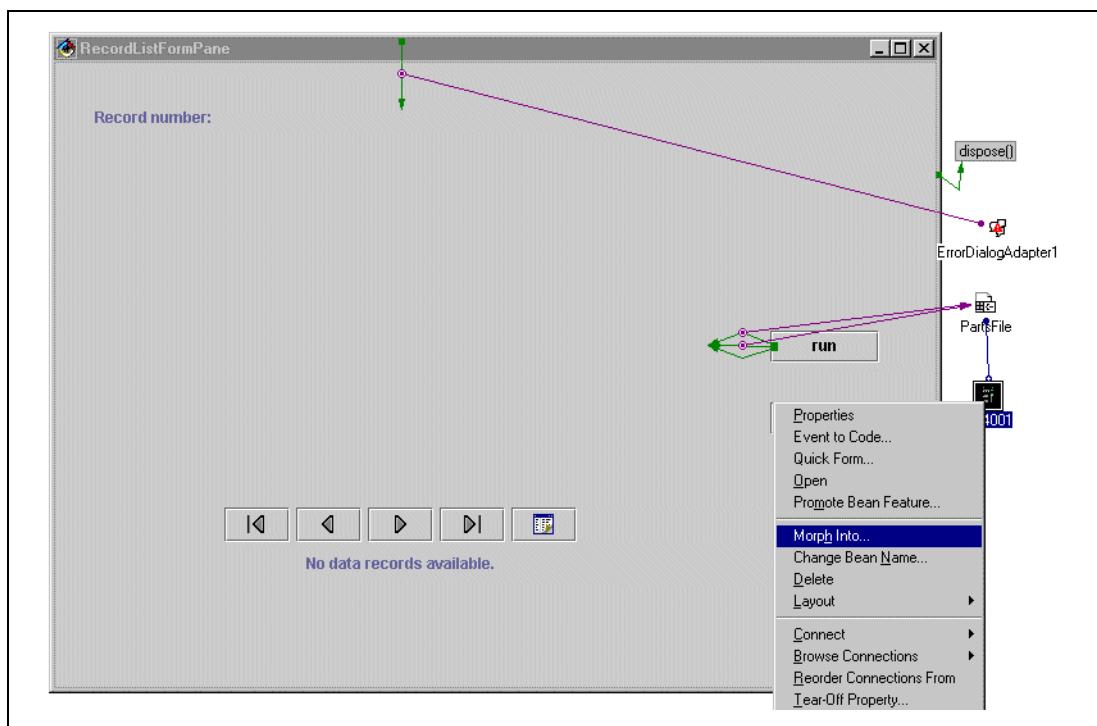


Figure 13-18 Morphing an AS400 object

3. In the Morph into dialog, enter:

```
com.ibm.as400.access.SecureAS400
```

Click **OK**. See Figure 13-19.

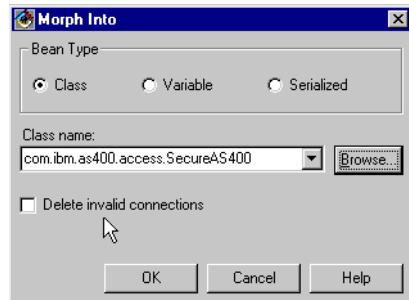


Figure 13-19 Morphing the AS400 object to a SecureAS400 object

4. Once the morph operation is completed, the AS400 object is morphed to a SecureAS400. It uses SSL as opposed to normal socket communications. You need to repeat this step for all AS400 objects used in your application.

### 13.9.3 Testing the changed program

When the client application is run, it will not indicate whether it is using SSL. However, if the program be unable to find the correct SSL-enabled server, a message appears through the ErrorDialogAdaptor, as illustrated in Figure 13-20.

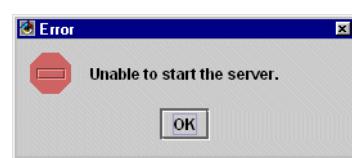


Figure 13-20 The ErrorDialogAdaptor message

If the application works and connects to the iSeries server, you can use the iSeries netstat \*cnn command to verify the type of socket communication that is being used.

#### 13.9.4 Additional SSL-related resources

For additional information, check out these resources:

- ▶ *HTTP Server Webmaster's Guide V4R5*, GC41-5434
- ▶ *AS/400 Internet Security: Protecting Your AS/400 from HARM in the Internet*, SG24-4929
- ▶ IBM Toolbox for Java and JTOpen: <http://www.iSeries.ibm.com/toolbox>
- ▶ iSeries Information Center:  
<http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm>
- ▶ IBM Web Application Servers site: <http://www.software.ibm.com/webservers/>
- ▶ IBM security site: <http://www.ibm.com/security>
- ▶ VeriSign Products and Services page: <http://www.verisign.com/products/doc.html>
- ▶ Netscape SSL-TLS search page:  
<http://directory.netscape.com/Computers/Internet/Protocols/SSL-TLS>
- ▶ RSA Security Inc. home page: <http://www.rsa.com>



A

# 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/SG246245>

Alternatively, you can go to the IBM Redbooks Web site at:

[ibm.com/redbooks](http://ibm.com/redbooks)

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246245.

## Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<i>File name</i>	<i>Description</i>
<b>apilib.sav</b>	iSeries V5R1 save file
<b>apilib45.sav</b>	iSeries V4R5 save file
<b>sg246245.jar</b>	VisualAge for Java code in a JAR file
<b>sg246245.dat</b>	VisualAge for Java code in a repository file
<b>readme.pdf</b>	Instructions for restoring the programming examples

## System requirements for downloading the Web material

The following system configuration is recommended:

**Hard disk space:** 10 MB is required for download material  
**Operating System:** Windows 98, NT or 2000

**Processor:** 300 Mhz or higher  
**Memory:** 128 MB

## 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. Read the file named readme.pdf to view information about how to restore the program code to VisualAge for Java and the iSeries server.

# Special notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere., The Power To Manage., Anything. Anywhere., TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

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

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

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

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

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

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

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.

# 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 534.

- ▶ *AS/400 Client/Server Performance Using the Windows Client*, SG24-4526
- ▶ *AS/400 Internet Security: Protecting Your AS/400 from HARM in the Internet*, SG24-4929
- ▶ *VisualAge for Java Enterprise Version 2 Team Support*, SG24-5245
- ▶ *Programming with VisualAge for Java Version 3.5*, SG24-5264
- ▶ *AS/400 XML in Action: PDML and PCML*, SG24-5959
- ▶ *How about Version 3.5? VisualAge for Java and WebSphere Studio Provide Great New Function*, SG24-6131
- ▶ *Version 3.5 Self Study Guide: VisualAge for Java and WebSphere Studio*, SG24-6136

## Other resources

These publications are also relevant as further information sources:

- ▶ *HTTP Server Webmaster's Guide V4R5*, GC41-5434
- ▶ *ILE RPG for AS/400 V5R1 Programmer's Guide*, SC09-2507
- ▶ *OS/400 Work Management*, SC41-5306
- ▶ *Performance Tools for iSeries*, SC41-5340
- ▶ *DB2 for AS/400 Database Programming V4R3*, SC41-5701
- ▶ *Distributed Database Programming*, SC41-5702
- ▶ *System API Reference V4R4*, SC41-5801
- ▶ Flanagan, David. *Java in a Nutshell: A Desktop Quick Reference*. O'Reilly & Associates, 1999 (ISBN 1-56592-487-8).
- ▶ Morgan, Bryan. *Java Developer's Reference*. Sams, 1996 (ISBN 1-57521-129-7).
- ▶ Taylor, David. *Object Technology: A Manager's Guide*. Addison-Wesley Publishing Co., 1997 (ISBN 0-20130-994-7).
- ▶ Englander, Robert. *Developing JavaBeans*. Sebastopol, CA, O'Reilly & Associates, 1997 (ISBN 1-56592-289-1).
- ▶ Lemay, Laura. *Java 1.1 Interactive Course*. The Waite Group, 1997 (ISBN 1-57169-083-2).
- ▶ Coulthard, Phillip and Farr, George. *Java for RPG Programmers*. IBM Press, 1998 (ISBN 1-88967-123-1) (GK2T-9890; available on diskette).
- ▶ Cooper, Alan. *About Face: The Essentials of User Interface Design*. Hungry Minds, Inc., 1995 (ISBN 1-56884-322-4).
- ▶ Galitz, Wilbert O. *User-Interface Screen Design*. John Wiley & Sons, 1993 (ISBN 0-47156-156-8).

- ▶ Mayhew, Deborah J. *Principles and Guidelines in Software User Interface Design*. Englewood Cliffs, New Jersey: Prentice Hall PTR/Sun Microsystems Press, 1997 (ISBN 0-13721-929-6).
- ▶ *Object-Oriented Interface Design* (IBM Common User Access™ Guidelines), Carmel, IN: Que, 1992.
- ▶ Preece, Jenny. *Human Computer Interaction*. Addison-Wesley Pub Co, 1994 (ISBN 0-20162-769-8).

## Referenced Web sites

These Web sites are also relevant as further information sources:

- ▶ IBM Toolbox for Java and JTOpen: <http://www.iseries.ibm.com/toolbox>
- ▶ iSeries Information Center:  
<http://publib.boulder.ibm.com/pubs/html/as400/infocenter.htm>
- ▶ IBM Web Application Servers site: <http://www.software.ibm.com/webservers/>
- ▶ IBM security site: <http://www.ibm.com/security>
- ▶ IBM and AS/400 Online Publications: <http://as400bks.rochester.ibm.com>
- ▶ VeriSign Products and Services page: <http://www.verisign.com/products/doc.html>
- ▶ VeriSign home page: <http://www.verisign.com>
- ▶ Netscape SSL-TLS search page:  
<http://directory.netscape.com/Computers/Internet/Protocols/SSL-TLS>
- ▶ RSA Security Inc. home page: <http://www.rsa.com>
- ▶ Java 2 SDK, Standard Edition documentation:  
<http://java.sun.com/products/jdk/1.2/docs>
- ▶ VisualAge Developer Domain for EJB: <http://www.ibm.com/software/vadd>
- ▶ Toolbox for Java JTOpen Web site:  
<http://oss.software.ibm.com/developerworksopensource/jt400>
- ▶ Java Foundation Classes Web site: <http://www.javasoft.com/products/jfc/index.html>
- ▶ Java White Papers site: <http://java.sun.com/docs/white/index.html>
- ▶ Java Native Interface page:  
<http://www.javasoft.com/products/jdk/1.2/docs/guide/jni/index.html>
- ▶ Oracle Technologies site for SQLJ and JDBC:  
<http://www.oracle.com/java/sqlj/index.html>
- ▶ *Trail: Java Native Interface* by Beth Stearns at:  
<http://java.sun.com/nav/read/Tutorial/native1.1/index.htm>

## How to get IBM Redbooks

Search for additional Redbooks or redpieces, view, download, or order hardcopy from the Redbooks Web Site

[ibm.com/redbooks](http://ibm.com/redbooks)

Also download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

## **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, updates and formats.



# List of Abbreviations

<b>AFP</b>	Advanced Function Printing	<b>IDE</b>	Integrated Development Environment
<b>ASP</b>	Active Server Pages	<b>IDL</b>	Interface Definition Language
<b>APA</b>	All Points Addressable	<b>IOP</b>	Internet inter-ORB protocol
<b>AWT</b>	Abstract Windowing Toolkit	<b>IP</b>	Internet Protocol
<b>CA</b>	Certificate Authority	<b>ITSO</b>	International Technical Support Organization
<b>CORBA</b>	Common Object Request Broker Architecture	<b>JAR</b>	Java archive
<b>COM</b>	Component Object Model	<b>JDBC</b>	Java Database Connectivity
<b>CGI</b>	Communications Gateway Interface	<b>JDK</b>	Java Development Toolkit
<b>CPW</b>	Commercial Processing Workload	<b>JFC</b>	Java Foundation Classes
<b>DAX</b>	Data Access Builder	<b>JIT</b>	Just in Time Compiler
<b>DBMS</b>	Database management system	<b>JPDC</b>	Java Performance Data Converter
<b>DCM</b>	Digital Certificate Manager	<b>JSP</b>	Java Server Pages
<b>DDM</b>	Distributed Data Management	<b>JVM</b>	Java virtual machine
<b>DLL</b>	Dynamic Link Library	<b>LDAP</b>	Lightweight Directory Access Protocol
<b>DPC</b>	Distributed Program Call	<b>MI</b>	Machine Interface
<b>EAB</b>	Enterprise Access Builder	<b>MIME</b>	Multi-purpose Internet Mail Extensions
<b>EJB</b>	Enterprise JavaBeans	<b>MVC</b>	Model View Controller
<b>FFST</b>	First Failure Support Technology	<b>NSAPI</b>	Netscape API
<b>GUI</b>	Graphical User Interface	<b>ODBC</b>	Open Database Connectivity
<b>GWAPI</b>	Go Web Server API	<b>OOA</b>	Object-oriented analysis
<b>HTML</b>	Hypertext Markup Language	<b>OOD</b>	Object-oriented design
<b>HTTP</b>	Hypertext Transmission Protocol	<b>OOP</b>	Object-oriented programming
<b>HTTPS</b>	Hypertext Transmission Protocol Secure	<b>ORB</b>	Object Request Broker
<b>IBM</b>	International Business Machines Corporation	<b>PEX</b>	Performance Explorer
<b>ICAPI</b>	IBM Connection API	<b>PCML</b>	Program Call Markup Language
		<b>PDM</b>	Panel Definition Markup Language
		<b>PTDV</b>	Performance Trace Data Visulizer

<b>PTF</b>	Program temporary fix
<b>RAD</b>	Rapid Application Development
<b>RAWT</b>	Remote Abstract Windowing Toolkit
<b>RMI</b>	Remote method invocation
<b>SCS</b>	SNA Character Set
<b>SLIC</b>	System Licensed Internal Code
<b>S-MIME</b>	Secure Multi-purpose Internet Mail Extensions
<b>SNA</b>	System Network Architecture
<b>SSL</b>	Secure Sockets Layer
<b>TCP</b>	Transmission Control Protocol
<b>TIMI</b>	Technology Independent Machine Interface
<b>UML</b>	Unified Methodology Language
<b>URL</b>	Universal Resource Locator
<b>VCE</b>	Visual Composition Editor
<b>WAS</b>	WebSphere Application Server
<b>WWW</b>	World Wide Web
<b>XML</b>	Extensible Markup Language

# Index

## Symbols

\*DEBUG option 479

## Numerics

5722-ACx 508  
5722-CEx 508  
5763-JC1 258  
5769-ACx 517  
5769-JC1 124

## A

absolute positioning 163  
Abstract Windowing Toolkit (AWT) 34  
access class 129, 152  
    additional 216  
accountability 510  
Add Environment Variable (ADDENVVAR) command 46, 368  
ADDENVVAR (Add Environment Variable) command 46, 368  
adding a new package 83  
adding item 337  
addOrderHeader() method 375, 380, 392, 432, 452  
addOrderLine() method 375, 381, 392, 433, 452  
addRecord method 159  
ajar tool 22  
API  
    Core Library 11  
    embeddedjava 13  
    Java Commerce 12  
    Java Management 13  
    javahelp 12  
    media and communications 12  
    personaljava 13  
    server 12  
    servlet 12  
appearance of a frame 89  
applet 110  
Applet Viewer 110  
appletviewer tool 15, 22  
application 7  
    closing 106  
    description 219  
    flow 301, 324  
    securing with SSL 506  
    version 98  
Application Programming Interface (API) 11  
    Core Library API 11  
    Enterprise API 11  
    Standard Extension Library API 11  
AS400 object 133  
AS400ToolboxInstaller class 498, 499  
AS400ToolboxJarMaker 130, 500, 503

example 506  
authentication 509  
authenticity 509  
AWT (Abstract Windowing Toolkit) 34

## B

base API package 134  
BeanInfo 78  
Blob 163  
breakpoint 103, 114, 480  
browser 67  
    class 75  
    package 74  
    project 71  
    type 75  
button function 93  
bytecode 9  
    interpreter 9  
    verifier 9

## C

C program  
    ChangeStgC 469  
    HelloC 467  
    INVOKEJAVA 472  
CA (certifying authority) 510  
certificate authority (CA) 506, 510, 511, 512, 513  
certifying authority (CA) 510  
Change Java Program (CHGJVAPGM) command 26  
    CLSF parameter 26  
    ENBPFRCOL parameter 26  
    LICOPT parameter 26  
    MERGE parameter 26  
    OPTIMIZE parameter 26  
ChangeStgC 469  
ChangeStgJ 468  
CHGJVAPGM (Change Java Program) command 26  
CHKPATH parameter 28  
class 3, 79, 85, 314  
    adding to a package 84  
    browser 75  
    loader 9  
    methods 317  
    relationships 4  
class loader 9  
CLASS parameter 28  
class variable 314, 317  
CLASSPATH 149, 355, 365, 368, 369  
    environment variable 46, 50  
    parameter 28  
recommendation 50  
setting in Qshell Interpreter 47  
setting on every Java CL command 46

setting the variable 46  
 setting up the variable 54  
 using the -classpath parameter 47  
 -classpath parameter 47  
 client 354, 365  
 client class 401  
 Clob 163  
 closing 106  
 CLSF parameter 23, 26  
 collaboration 5  
 command  
     Change Java Program (CHGJVAPGM) 26  
     Create Java Program (CRTJVAPGM) 23, 479  
     CRTJVAPGM (Create Java Program) 479  
     Delete Java Program (DLTJVAPGM) 27, 479  
     Display Java Program (DSPJVAPGM) 27  
     Display Object Description (DSPOBJD) 376  
     DLTJVAPGM (Delete Java Program) 479  
     Monitor Message (MONMSG) 376  
     Run Java (RUNJVA) 28  
     Start Debug (STRDBG) 486  
     Start Debug Server (STRDBGSVR) command 489  
     Start Qshell (STRQSH) 32  
     Start Service Job (STRSRVJOB) 486  
     Submit Job (SBMJOB) 472  
     Work with Active Jobs (WRKACTJOB) 486  
 Command call class 146  
 command call GUI components 221  
 command, CL  
     Monitor Message (MONMSG) 376  
     MONMSG (Monitor Message) 376  
     Start Debug (STRDBG) 486  
     Start Service Job (STRSRVJOB) 486  
     STRDBG (Start Debug) 486  
     STRSRVJOB (Start Service Job) 486  
 Commerce API 12  
 commitOrder() method 375, 379, 391, 432, 452  
 comparing SQLJ and JDBC 407  
 compile support 295  
 compiling Java on the iSeries 319  
 component 7  
     browser 70  
     hierarchy 68  
     miscellaneous 10  
 composition 5  
 confidentiality 509  
 connect method 207, 214  
 connection pool 166  
 connectToDB method 154, 174, 180, 188, 198  
 console 118  
 constructor method 161  
 Convert Display File SmartGuide 268  
 converting the OrderEntryWdw2 class to use SQLJ 423  
 converting the SltCustWdw program to use SQLJ 424  
 cooperative debugger 489, 495  
 CORBA 350  
 Core Library API 11, 19  
     java.applet 11  
     java.awt 11  
     java.beans 11  
     java.io 11  
     java.lang 11  
     java.math 11  
     java.net 11  
     java.rmi 11  
     java.security 11  
     java.sql 11  
     java.text 11  
     java.util 11  
 Create Java Program (CRTJVAPGM) command 23, 295, 319, 479  
     CLSF parameter 23  
     ENBPFRCOL parameter 25  
     LICOPT parameter 25  
     OPTIMIZE parameter 24  
     REPLACE parameter 24  
     SUBTREE parameter 25  
     USEADPAUT parameter 24  
     USRPRF parameter 24  
 Create Program Call SmartGuide 259  
 CRTJVAPGM (Create Java Program) command 23, 295, 489  
 CstmrlInq example 408  
     running 411  
     setting up VisualAge for Java 408  
 CstmrlInqJ 448  
 CSTMRINQR 446  
 CstmrlList example 417, 418  
     running 417  
     setting up VisualAge for Java 417  
 customer list 328, 344  
 Customer Table Layout (CSTMRL) 308  
 customer transaction 300  
     flow 302

## D

data conversion 135  
 data description 135  
 Data File Utility (DFU) beans 279  
 data queue 194, 221  
     application example 196  
     class 147  
     DataQueue object 199  
     read 201, 204  
     write 201, 204  
 database connection 325  
 database management system (DBMS) 36  
 database table structure 307  
 database terminology 311  
 DataQueueExample class 198  
 DBMS (database management system) 36  
 DDM server 145, 179  
 DDM-record level access 372  
 DDS 271  
 debug, starting 486  
 debugger 112, 115  
     OS/400 479  
     stepping through methods 116  
 debugging 103  
     \*DEBUG option 479

breakpoints 480  
compiling code to debug 478  
display variable 482  
from another terminal session 486  
iSeries Java program 492  
preparations 478  
run 495  
setting breakpoints 480  
step debug 495  
step over 495  
debugging programs 477  
DefaultContext 408  
Delete Java Program (DLTJVAPGM) command 27, 479  
deleteRecord method 159, 194, 205  
deployment considerations and tools 497  
DFU beans 279  
example 280  
differences between Java and RPG 315  
digital certificate 127, 508, 510, 511  
environment 513  
Digital Certificate Management (DCM) 512  
Digital Certificate Manager (DCM) 506, 512, 513, 517, 518, 519  
digital signature 509  
directory authority 512  
Display Installed Licensed Programs display 40  
Display Java Program (DSPJVAPGM) command 27  
    OUTPUT parameter 27  
Display Object Description (DSPOBJD) command 376  
displaying variables 482  
dispose method 159, 178, 184, 194, 205  
distributed data management (DDM) record-level access 179  
Distributed Debugger 489  
    controlling a session 495  
Distributed Program Call (DPC) 186, 328, 341  
Distributed Program Call (DPC) feature 185  
Distributed Program Call feature 259  
Distributed Program Call SmartGuide 259  
District Table Layout (Dstrct) 308  
DistrictJ 443  
DISTRICTR 444  
DLTJVAPGM (Delete Java Program) command 27, 479  
DPC (Distributed Program Call) 186, 328, 341  
    application example 186  
    ProgramCall object 188  
    ProgramParameter 189  
DPCExample class 187  
DPCXRPG bean 265  
DPCXRPG program name 187  
driver manager 228  
DSPJVAPGM (Display Java Program) command 27  
DSPOBJD (Display Object Description) command 376  
dynamic binding 6

## E

EAB (Enterprise Access Builder) 66  
edition 109  
editor 112  
editor pane 114

EmbeddedJava API 13  
ENBPFRCOL parameter 25, 26  
encapsulation 3  
encryption 509  
Enterprise Access Builders (EAB) 66  
Enterprise API 11  
error event 222  
error handling 227  
error processing 151  
error recovery 151  
ET/400 257  
    compile support 295  
    Convert Display File SmartGuide 268  
    Create Program Call SmartGuide 259  
    create, run, and debug 293  
    export 293  
    export support 295  
    IBM Toolbox for Java classes 258  
    run support 296  
    setup 293  
    system requirements 297  
    using 258  
EVAL function 484  
event-to-method connection 81  
exception, throwing 376  
export 293, 364  
exporting Java files 295  
extending the application 99  
External Version Control 107

## F

FFDC (First Failure Support Technology) 151  
field description object 202  
final variables 314  
finalize() method 436, 454  
First Failure Support Technology (FFDC) 151  
formatSpooledFile method 208  
frame appearance 89  
framework 7  
FTP classes 216  
FTP subclass 217

## G

garbage collector 10  
GCFRQ parameter 29  
GCHINL parameter 29  
GCHMAX parameter 29  
GCPTY parameter 29  
getCustomerDiscount() method 375, 382, 393, 434, 453  
getOrderNumber() method 375, 383, 394, 435, 453  
getRawDate() method 386, 396, 436, 454  
getRawTime() method 386, 396, 436, 454  
getRecord method 156, 175, 182, 188, 199  
getSpooledFilesForUser (String User) method 209  
getters 5  
global fields 314  
Graphical Toolbox 131  
graphical user interface (GUI) 321

Group class 127  
groups 224  
GUI (graphical user interface) 321  
GUI classes 127, 219  
    command call 221  
    data queues 221  
    error events 222  
    iSeries panes 219  
    JDBC 220  
    JDBC examples 226  
    jobs 222  
    keyed access example 240  
    messages 223  
    network print 223  
    overview 219  
    program call 223  
    record-level access 224  
    RecordListFormPane example 238  
    SQL connection 221  
    SQLQueryBuilderPane example 230  
    SQLResultSetFormPane example 233  
    SQLResultSetModel example 236  
    users and groups 224  
GUI component classes 242  
GUI examples 238

## H

hashing 509  
Hello C example 466  
Hello World 118  
HelloC 467  
HelloJ 466  
host server 132, 133, 149  
HTML 112  
HTTP server 149  
HTTPS 508

## I

IBM Cryptographic Access Provider 517  
IBM Developer Kit for Java (5722-JV1) 41  
IBM Distributed Debugger 489  
IBM Enterprise Toolkit for AS/400 (ET/400) 257  
IBM Enterprise Toolkit for AS400 project 124, 258  
IBM Native JDBC driver 361  
IBM Toolbox for Java 120, 123  
    certificate 522  
    classes 258  
    data conversion 135  
    digital certificates 127  
    enhancements 127  
    GUI classes 127, 219  
    host servers 132  
    installation and update 498  
    installing 124  
    installing on the workstation 53  
    introduction 124  
    Jobs class 127  
    Message Queue class 127  
    proxy support 148, 252

QueuedMessage class 127  
security 150  
servlet support 148  
User and Group class 127  
UserSpace class 128  
V4R3 enhancements 127  
V4R4 enhancements 128  
V4R5 enhancements 130  
V5R1 enhancements 131  
IBM Toolbox for Java (5769-JC1) 41, 323  
IBM Toolbox for Java classes 258  
    copying to the workstation 56  
    in the VCE 226  
IBM Toolbox for Java Modification 4 125  
IDE (Integrated Development Environment) 65, 66  
IDE setup 114  
identities 15  
IDL (Interface Definition Language) 12  
IFS  
    available 216  
    connectService 214  
    IFSFile 215  
    IFSInputStream 216  
    list 215  
    read 216  
IFSEExample class 214  
IFSFileDialog 224  
IFSJavaFile class 210  
IFSTextFileDocument 225  
inheritance 318  
Initial Program to Call (INLPGM) 50  
initialize() method 374, 376, 390, 431, 452  
initRecordFormat method 201  
inspectors 116  
install  
    Java on the iSeries server 38  
    software 39  
instance method 318  
instance variable 161, 174, 180, 187, 198, 314, 318  
instances 3  
Integrated Development Environment (IDE) 65, 66  
integrated file system 145, 224, 293, 364  
    example 211  
integrated file system access 210  
integrity 509  
interface 351, 359  
Interface Definition Language (IDL) 12  
Internet CA 518  
Internet security elements 507  
intranet CA 517  
intranet certificate authority 513  
Invocation API 446  
INVOKEJAVA 472  
iSeries application, building with RMI 352  
iSeries beans 271  
iSeries data type 135, 151  
iSeries database access 152  
iSeries implementation 8  
iSeries implementation of Java 1  
iSeries Java configuration 37

iSeries Java virtual machine 17  
iSeries native compile 478  
iSeries pane 224  
iSeries panes 219  
    AS400DetailsPane 220  
    AS400ExplorerPane 220  
    AS400ListPane 219  
    AS400TreePane 220  
iSeries server  
    debugging a Java program 492  
    debugging Java programs 477  
    implementing DCM 512  
    Java APIs 19  
    Java utilities 19  
    required software 38  
    running Cstmrlnq 412  
    running Remote AWT 59  
    SQLJ 408  
    testing the Java environment 50  
iSeries Universal Database 163  
iSeries-specific implementation 22  
    java command 23  
Item class 358  
item list 333  
Item Table Layout (ITEM) 310  
ItemDetail 359  
ItemEntryl 359, 360  
iterator 417, 419

## J

JAR (Java ARchive) file 9  
jar tool 15  
JarMaker 130, 500, 501  
    class 500  
    example 502  
Java 315  
    Application Programming Interface (API) 11  
    compiler 14  
    compiling on the iSeries server 319  
    configuration on iSeries 37  
    debugger 14  
    debugging 477  
    debugging on the iSeries server 492  
    environment testing on the iSeries server 50  
    exporting files 295  
    for RPG programmers 313, 315  
    moving the server application 349  
    object 3  
    on the iSeries server 16  
    overview 1, 8, 63, 123, 257, 313  
    platform 8  
    programming language 2  
    setting up the environment for CL commands 45  
    syntax 316  
    utilities 13  
Java APIs 11, 19  
Java ARchive (JAR) file 9  
Java archive file 500  
Java class  
    ChangeStgJ 468

CstmrlnqJ 448  
DistrictJ 443  
HelloJ 466  
Java client 321  
Java client GUI 323  
JAVA command 28  
java command 13, 20, 23, 28, 413  
Java commands  
    Change Java Program (CHGJVAPGM) 26  
    Create Java Program (CRTJVAPGM) command 23  
    Delete Java Program (DLTVAPGM) command 27  
    Display Java Program (DSPJVAPGM) command 27  
    Run Java (RUNJAVA) command 28  
Java data type 151  
Java Database Connectivity (JDBC) 66, 220  
Java Development Kit (JDK) 8  
Java Foundation Classes (JFC) 12  
Java interface 352  
Java Invocation API 471, 472  
Java Management API 13  
Java Media and Communications API 12  
Java method 101, 446  
    prototyping in RPG 455  
Java Naming and Directory Interface (JNDI) 12, 163  
Java Native Interface (JNI) 10, 439  
    C 464  
    C header files 464  
    C, setting up 464  
    calling an RPG procedure from a Java program 443  
    introduction 440  
    Java Invocation API 472  
    Java objects 446  
    Order Entry application 450  
    positioning 440  
    RPG 442  
    who should use it 441  
Java Native Interface (JNI) and C  
    changing a Java String object 468  
    method calls 466  
Java Native Interface (JNI) and RPG requirements 442  
Java package  
    java.applet 11  
    java.awt 11  
    java.beans 11  
    java.io 11  
    java.lang 11  
    java.math 11  
    java.net 11  
    java.rmi 11  
    java.security 11  
    java.sql 11  
    java.text 11  
    java.util 11  
Java program, debugging 477  
Java Servlet API 12  
Java String object 468  
Java Transformer 23  
Java utilities 19  
    ajar 22  
    appletviewer 15, 22

jar 15  
 java 13, 20  
 javac 14, 21  
 javadoc 15  
 javah 14  
 javakey 15  
 javap 14, 22  
 jdb 14  
 native2ascii 16  
 rmic 15  
 rmiregistry 16  
 serialver 16  
 Java virtual machine (JVM) 9  
 java.applet package 11  
 java.awt API 19  
 java.awt package 11  
 java.beans package 11  
 java.io API 19  
 java.io package 11  
 java.lang package 11  
 java.math package 11  
 java.net API 19  
 java.net package 11  
 java.rmi package 11  
 java.security package 11  
 java.sql 11  
 java.sql API 19  
 java.sql package 11  
 java.text package 11  
 java.util package 11  
 JavaApplicationCall class 217  
 JavaBeans 79  
 javac 21, 319, 478  
 javac command 14  
 javadoc command 15  
 javah command 14, 21  
 JavaHelp API 12  
 javakey tool 15  
 javap command 14, 22  
 jdb command 14  
 JDBC 12, 329, 333, 361, 372, 388, 406, 408, 417
 

- application example 153
- CallableStatement object 171
- connection object 155
- example with RMI 355
- examples 226
- executeQuery 157, 176
- executeUpdate 159
- extended dynamic 139
- getConnection 155
- interface 138
- next() method 157
- Order Entry application 388
- package cache 139
- performance tips 139
- prepareCall 171, 175
- prepareStatement 155
- properties 139, 140
- ResultSet 157, 176
- stored procedure application example 172

stored procedures 171  
 supported interfaces 137  
 versions 137  
 JDBC (Java Database Connectivity) 66, 220  
 JDBC 2.0 129, 164
 

- absolute 169, 170
- beforeFirst 170
- CONCUR\_READ\_ONLY 168
- Concur\_Updatable 168
- example 167
- first 165, 170
- isLast 165
- last 165
- next 165
- previous 165
- scrollable result set 164
- scroll-insensitive 164
- scroll-sensitive 164

JDBC 2.0 Core API 163  
 JDBC 2.0 Standard Extension API 163  
 JDBC DataSource 166  
 JDBC initialize() 390  
 JDBC result sets 163  
 JDBC specification 137  
 JDBCExample class 154  
 JDBCExampleDisplayAll class 161  
 JDBC-ODBC bridge 372  
 JDBCRemote package 357  
 JDK (Java Development Kit) 8  
 JDK 1.1 66  
 JFormatted beans 271
 

- example 272

JNDI (Java Naming and Directory Interface) 12, 163  
 JNI (Java Native Interface) 10, 439  
 JNI and RPG consideration 450  
 jobs 127, 222  
 JTOpen 124  
 Just-In-Time (JIT) compiler 10, 139  
 JVM (Java virtual machine) 9, 17

## K

keyed access 224  
 keyed access example 240  
 keyed data queue 195  
 KeyRing class 524, 525

## L

LICOPT parameter 25, 26  
 local fields 314  
 local variables 314  
 log 118

## M

mapping RPG to Java 372  
 MERGE parameter 26  
 message list object 223  
 message queues object 223  
 message, monitoring 376

MessageQueue class 127  
messages 223  
method logic 376, 390  
method signature 475  
Method Source pane 114  
methods 314  
miscellaneous components 10  
modifications 127  
module list 485  
Monitor Message (MONMSG) command 376  
monitoring, message 376  
MONMSG (Monitor Message) command 376  
multiple JDK versions 60

## N

named constants 314  
national language support 150  
native compile on iSeries 478  
Native JDBC driver 361  
native2ascii tool 16  
network print 223  
network print class 206  
network print example 206  
new package 83  
non-keyed data queue 196

## O

object 3  
    encapsulation 3  
object creation 317  
object destruction 318  
object-oriented language 2  
object-oriented programming 8, 313, 314  
    classes 314  
    variables 314  
object-oriented technology benefits 6  
ODBC stored procedure 328  
optimization levels 296  
OPTIMIZE parameter 24, 26, 29  
OPTION parameter 29  
Oracle Corporation 408  
Oracle Reference Interpreter (RI) 408  
Order Entry application 299, 321, 323, 421, 428  
    changing the host 344  
        processing the submitted order 346  
        providing a customer list 344  
        providing an item list 345  
        verifying an item 346  
    cleaning up 386  
    creating Java on iSeries 371  
    database layout 307  
    DDM  
        cleaning up 386  
        method logic 376  
    Java Native Interface 451  
    JDBC 388  
        cleaning up 397  
        method logic 390  
    record-level access (DDM) 373

    replacing the Java code 371  
Order Entry client application 421  
Order Entry server application 370, 428  
Order Entry window  
    application flow 324  
    connecting to the database 325  
    overview 323  
    retrieving the customer list 328  
    retrieving the item list 333  
    submitting the order 338  
    verifying the item 337  
Order Line Table Layout (ORDLIN) 309  
Order object 338  
order submit 338  
Order Table Layout (ORDERS) 309  
OrderDetail 339  
OrderEntryWdw class 423  
OS/400 Host Servers (5769-SS1 Option 12) 41  
OS/400 Java commands 23  
OS/400 system debugger 477, 479  
OUTPUT parameter 27  
overloading methods 318  
overriding methods 318

## P

package 84  
    adding a new class 84  
    adding new 83  
package browser 74  
packages 69, 315  
packet sniffer 509  
Panel Definition Markup Language (PDML) 129  
parameter connection 81  
PARM parameter 28  
PARTS file 152  
PartsContainer interface 170  
PCML (Program Call Markup Language) 129  
PDML (Panel Definition Markup Language) 129  
PersonalJava API 13  
polymorphism 6  
populateAllParts method 161, 177, 183, 191, 203  
populateTable method 214  
PreparedStatement 362  
print  
    connectService 208  
    openSynchronously 209  
    setUserFilter 209  
    size 209  
    SpooledFileList 209  
print objects 146  
Program call class 147  
program call GUI components 223  
Program Call JavaBean 261  
Program Call Markup Language (PCML) 129  
program interface 328  
project browser 71  
projects 69  
PROP parameter 29  
property-to-property connection 81  
proxy server, classes enabled to work 253

proxy support 148, 252  
example 253

## **Q**

qsh command 19, 32  
Qshell Interpreter 19, 32, 413  
  setting CLASSPATH 47  
  setting up for entire iSeries server 49  
  setting up for individual users 48  
  setting up the environment 47  
  starting 32  
Qshell Interpreter (5769-SS1 Option 30) 41  
Qshell Utilities for AS/400 (5799-XEH) 41  
QueuedMessage class 127  
QUTCOFFSET system value 52

## **R**

readFile() method 215  
receiver 5  
RecordFormat object 202  
record-level access 145, 224, 371  
  application example 179  
  GUI examples 238  
record-level access (DDM) 372, 373  
record-level conversion 136  
RecordListFormPane 224, 238, 240  
RecordListModel 224  
RecordListTablePane 224  
re-designing the application 322  
reflection 10  
relative positioning 163  
Remote Abstract Windowing Toolkit (AWT) 34, 57  
  running RAWT 59  
Remote AWT  
  running on the iSeries server 59  
  starting on the workstation 58  
remote implementation class 353  
remote method invocation (RMI) 12, 349, 370, 398, 450  
  support 397  
Remote Procedure Call (RPC) 350  
remote server object 352, 359  
REPLACE parameter 24  
repository 108  
Repository Explorer 118  
requesting a server certificate 518  
required software 38  
resize handles 86, 92  
ResourceList classes 225  
ResourceListDetailsPane 225  
ResourceListPane 225  
resources 69  
reusable GUI part 170  
RLAExample class 180  
RMI  
  advantages 403  
  architecture 350  
  building an RMI application 351  
  five-step process 351  
  JDBC example 355

JDBC example program interface 357  
Naming.rebind method 353  
proxy 350  
 RemoteException 353  
skeleton 351, 363  
specification 350  
stub 350, 351, 363  
UnicastRemoteObject 352, 360  
URL 353  
RMI (remote method invocation) 12, 349, 370, 398  
RMI application design 398  
RMI registry 355, 368  
RMI support  
  adding to a client 400  
  adding to a server class 399  
rmic 351, 353, 363, 400  
rmic command 15  
rmiregistry 355, 368, 400  
rmiregistry command 16  
RPC (Remote Procedure Call) 350  
RPG and Java 315  
RPG and JNI consideration 450  
RPG application flow 301  
RPG native method example 443  
RPG Order Entry application 299  
  application flow 301  
  components 301  
  customer transaction 300  
  customer transaction flow 302  
  database 300  
  database table structure 307  
  overview 300  
  starting the application 302  
  tables 308  
RPG Order Entry example 450  
RPG program  
  CSTMRLNQ 446  
RPG program background 260  
RPG programmers and Java 313  
RPG service program 451, 455  
  DISTRICTR 444  
RPG subprocedure 458  
Run Java (RUNJAVA) command 28, 319  
  CHKPATH parameter 28  
  CLASS parameter 28  
  CLASSPATH parameter 28  
  GCFRQ parameter 29  
  GCHINL parameter 29  
  GCHMAX parameter 29  
  GCPTY parameter 29  
  OPTIMIZE parameter 29  
  OPTION parameter 29  
  PARM parameter 28  
  PROP parameter 29  
run support 296  
RUNJAVA (Run Java) command 28  
running Cstmrlnq 411

## **S**

sample application 82

save and restore 151  
 SBMJOB (Submit Job) command 472  
 SCM (Software Configuration Management) 107  
 Scrapbook 118  
 scroll sensitive 168  
 scrollable result set 163, 164  
 Secure Sockets Layer (SSL) 508, 517  
 SecureAS400 526  
 security 150  
     application communication with SSL 506  
     Internet 507  
     transaction 508  
 self-signed certificate 513, 522  
 sender 5  
 serialization 10  
 serialver command 16  
 Server API 12  
 server application 349  
 server certificate 517, 518  
 server code, network accessible 355, 368  
 service job, starting 486  
 servlet support 148  
 setFetchSize method 165  
 setters 5  
 setting breakpoints 480  
 setting up Remote AWT 57  
 shared repository 107  
 SltCustWdw class 424  
 SltItemWdw class conversion 427  
 SmartGuide 82, 119  
     applet creation 110  
 sockets 350  
 software 38  
 Software Configuration Management (SCM) 107  
 solutions 69  
 specialization 4  
 spooled file example 207  
 SpooledFileListExample class 207  
 SpooledFileViewer 243  
 SpooledFileViewer object 223  
 SQL connection 221, 324  
 SQL INSERT 392  
 SQL stored procedure 344  
 SQLJ  
     #sql 407  
     considerations 437  
     runtime.zip 408  
     translator 407  
 SQLJ (Structured Query Language for Java) 405  
 SQLJ and JDBC 407  
 SQLJ and the iSeries server 408  
 SQLJ translator 408  
 SQLQueryBuilderPane 221, 230  
 SQLResultSetFormPane 221, 233  
 SQLResultSetModel 236  
 SQLResultSetTableModel 221  
 SQLResultSetTablePane 221, 226  
 SQLResultSetTablePane application 226  
 SQLStatementButton 221  
 SQLStatementDocument 221  
 SQLStatementMenuItem 221  
 SSL 506, 508, 511  
     self-signed certificate 513  
     to secure applications 506  
     with VisualAge 3.5 524  
 SSL (Secure Sockets Layer) 129  
 sslightu.zip 525  
 sslightx.zip 525  
 SSLTools.zip 525  
 Standard Extension API 11  
 Standard Extension Library 12  
     Commerce API 12  
     EmbeddedJava API 13  
     Java Management API 13  
     JavaHelp API 12  
     Media and Communications API 12  
     PersonalJava API 13  
     Server API 12  
     Servlet API 12  
 Start Debug (STRDBG) command 486  
 Start Debug Server (STRDBGSVR) command 489  
 Start Qshell (STRQSH) command 32  
 Start Service Job (STRSRVJOB) command 486  
 starting  
     debug 486  
     Qshell Interpreter 32  
     service job 486  
 Stock Table Layout (Stock) 310  
 stored procedure 171, 331  
 stored procedure application example 172  
 StoredProcedureExample class 174  
 STRDBG (Start Debug) command 486  
 STRDBGSVR (Start Debug Server) command 489  
 STRQSH (Start Qshell) command 32  
 STRSRVJOB (Start Service Job) command 486  
 Structured Query Language for Java (SQLJ) 405  
     CstmrInq explanation 414  
     CstmrList example 418  
     example  
         Order Entry application 423, 424  
     introduction 406  
     Order Entry application 421  
     Order Entry server application 428  
         converting the host program to use SQLJ 430  
         setting up 429  
     overview 407  
     retrieving a group of records from the customer table  
         using SQLJ 417  
 subclass 318  
 subfile 271  
 Submit Job (SBMJOB) command 472  
 SUBTREE parameter 25  
 Swing 219, 242  
 symbolic link 45  
 syntax for Java 316  
 System Licensed Internal Code (SLIC) 16  
 system status 225  
 system value GUI 226

## T

tables 308  
TCP/IP 508  
TCP/IP Connectivity Utilities for AS/400 (5769-TC1) 41  
team development 106  
Team01Project 83  
Technology Independent Machine Interface (TIMI) 17  
text component 222  
Thawte 511, 518  
thread 19, 318  
throwing an exception 376  
TIMI (Technology Independent Machine Interface) 17  
Toolbox for Java 41, 120, 123, 323  
    certificate 522  
    classes 258  
    data conversion 135  
    digital certificates 127  
    enhancements 127  
    GUI classes 127, 219  
    installation and update 498  
    installing 124  
    installing on the workstation 53  
    introduction 124  
    Jobs class 127  
    Message Queue class 127  
    proxy support 148, 252  
    QueuedMessage class 127  
    security 150  
    servlet support 148  
    User and Group class 127  
    UserSpace class 128  
    V4R3 enhancements 127  
    V4R4 enhancements 128  
    V4R5 enhancements 130  
    V5R1 enhancements 131  
Toolbox for Java classes 258  
    in the VCE 226  
transaction security 508  
translator.zip 408  
trusted authority 522  
trusted root 510  
trusted root key 510

## U

Universal Database (UDB) 163  
updatable result sets 163, 164  
updateCustomer() method 376, 384, 395, 435, 454  
updateRecord method 157, 192, 204  
updateStock() method 376, 384, 396, 435, 454  
URL 353, 361  
USEADPAUT parameter 24  
User class 127  
users 224  
UserSpace class 128  
USRPRF parameter 24

## V

V4R3 enhancements 127  
V4R4 enhancements 128

V4R5 enhancements 130  
V5R1 enhancements 131  
variable 314, 482  
variables/member variables/data members 3  
VeriSign 511, 518  
version 98, 106, 109  
versioning 106  
VIFSDirectory 225  
visual builder connection 81  
visual component 86  
Visual Composition Editor 77, 78, 81  
    connections 81  
    free-form surface 85  
    parts palette 85  
    resize-handles 92  
    toolbar 92  
VisualAge for Java 63, 323, 400  
    Applet Viewer 110  
    building a sample application 82  
    component hierarchy 68  
    debugger 115  
    debugging code 103  
    Enterprise Edition 65, 108  
    Entry Edition 64  
    help 119  
    inspector 116  
    Integrated Development Environment 65  
    Java Applet Viewer 110  
    modifying an application to use SSL 524  
    other windows 118  
    overview 64  
    prerequisites 119  
    Professional Edition 64, 108  
    Professional Edition with WebSphere Development Studio for iSeries 65  
    running Cstmrlnq 411  
    scrapbook 118  
    setting breakpoints 103  
    SmartGuide 112  
    starting 66  
    system requirement 119  
    versions 64  
VisualAge for Java 3.5 compile 478  
VisualAge for Java Debugger 103  
VisualAge for Java Help 119  
VJobList 250  
VPrinter object 223  
VPrinterOutput object 223  
VPrinters object 223  
VSystemStatusPane 249  
VUserAndGroup 224, 251  
VUserList 224, 251

**W**  
wizards 119  
Work with Active Jobs (WRKACTJOB) command 486  
Work with Environment Variable (WRKENVVAR) command 46  
Work with Licensed Programs menu 39  
Work with Module List display 485

Workbench 67  
Workbench window 69  
workspace 67  
writeDataQueue() method 376, 385, 396, 436, 454  
writeOrderLine() method 454  
WRKACTJOB (Work with Active Jobs) command 486  
WRKENVVAR (Work with Environment Variable) command 46





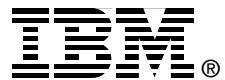
**Redbooks**

# Building Java Applications for the iSeries Server with VisualAge for Java 3.5

(1.0" spine)  
0.875" <-> 1.498"  
460 <-> 788 pages







# Building Java Applications for the iSeries Server

## with VisualAge for Java 3.5



### Install and configure Java for the iSeries server environment

In the past several years, Java has become the hot new programming language. The reasons for Java's popularity are its portability, robustness, and ability to produce Internet-enabled applications.

### Use the IBM Toolbox for Java to build iSeries Java applications

This IBM Redbook explains how you can use Java and the IBM *e*Server iSeries server to build server applications and client/server applications for the new network computing paradigm. It focuses on two key products: VisualAge for Java Version 3.5 and IBM Toolbox for Java.

### Use VisualAge for Java iSeries unique support

Throughout this Redbook, you'll find many practical programming examples with detailed explanations on how they work. You'll see how to modernize legacy RPG applications in a practical and evolutionary way through client and server Java examples. These examples are available for download from the Redbooks Web site. To understand this code better, download the files, as explained in Appendix A, and use them as a reference.

This Redbook is intended to help customers and service providers who need to install and configure Java on the iSeries server. It also targets application developers who want to develop Java applications for the iSeries server. Reading this Redbook will help you to quickly and easily start using Java with the iSeries server.

### **INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION**

### **BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:**  
[ibm.com/redbooks](http://ibm.com/redbooks)